Defence Research and Development Canada
Recherche et développement pour la défense Canada

Practical verification & safeguard tools for C/C++

F. Michaud
R. Carbone
DRDC Valcartier

Defence R&D Canada – Valcartier
Technical Report
DRDC Valcartier TR 2006-735
November 2007

Canada

Practical verification & safeguard tools for C/C++

F. Michaud
R. Carbone
DRDC Valcartier

DRDC Valcartier

Technical Report

DRDC Valcartier TR 2006-735

November 2007

Principal Author

Approved by

Yves van Chestein
Head/IKM

Approved for release by

Christian Carrier
Chief Scientist

Abstract

This document is the final report of an activity that took place in 2005-2006. The goal of this project was first to identify common software defects related to the use of the C and C++ programming languages. Errors and vulnerabilities created by these defects were also investigated, so that meaningful test cases could be created for the evaluation of best-of-breed automatic verification tools. Finally, when relevant, best practices were inferred from our experiments with these tools.

Résumé

Ce document est le rapport final d'un projet de recherche qui a eu lieu en 2005-2006. Le but de ce projet était avant tout d'identifier les défauts logiciels courants liés à l'utilisation des langages de programmation C et C++. Les erreurs et vulnérabilités créées par ces défauts ont aussi été étudiées, de manière à rendre possble la création de tests significatifs pour l'évaluation des meilleurs outils de vérification disponibles. Finalement, lorsque pertinent, de bonnes pratiques ont été déduites de nos expérimentations avec ces outils.

This page intentionally left blank.

Executive summary

Practical verification & safeguard tools for C/C++

F. Michaud, R. Carbone; DRDC Valcartier TR 2006-735; DRDC Valcartier; November 2007.

This document is the final report of an activity that took place in 2005-2006. The goal of this project was first to identify common software defects related to the use of the C and C++ programming languages. Errors and vulnerabilities created by these defects were also investigated, so that meaningful test cases could be created for the evaluation of best-of-breed automatic verification tools. Finally, when relevant, best practices were inferred from our experiments with these tools. Since our team had already investigated design problems in 2002-2005, this study of implementation (or programming) problems was the next logical step.

We found that C and C++ are both very prone to defects, errors and vulnerabilities with serious consequences, such as arbitrary code execution. This is mostly due to three factors. First, C and C++ give too much control to the programmer over low-level and complex things that are easily misused, such as memory management and pointer arithmetic. In other words, they give programmers enough rope to hang themselves. Second, the execution of a C/C++ program is not supervised by a monitor and, because of this, many errors such as buffer overflows are not detected until it is too late. Finally, many aspects of these languages are badly designed (e.g. static buffers and null-terminated strings) and tend to increase the impacts of other problems.

We reviewed over 30 tools that were split into five categories and came mostly from two worlds: academic/open-source and proprietary/commercial. We found that the commercial tools were generally a lot better than the free ones. This was expected, but not to the level where most free tools are worthless and commercial ones excellent, since in the Java world (as an example) there are many excellent free verification tools. We believe this is because correctly parsing C/C++ programs is a huge challenge and this creates a barrier to entry that only well-financed (commercial) projects can overcome.

Of all the tools we reviewed, only five had the potential to be really useful and were further evaluated. To do so, two series of tests were created: synthetic tests and production code tests. Synthetic tests consisted of 25 C++ classes containing defects, all integrated into a high quality open-source application. All five tools performed well and to a similar degree on synthetic tests: together, they found all but four problems. Production code tests used a small numerical analysis application of about 10K lines that was known to be a bit buggy. This time, many tools had a lot of difficulties and found almost nothing. It seems that low-quality 'spaghetti' code that is using pointers heavily can be a showstopper for many tools, but few others were able to analyze the application and detect problems easily.

Our main recommendation is to use better languages and platforms, such as Java, C#, Ada, or any other modern programming language. Of course, this is not always possible for many

reasons and using verification tools can be an interesting alternative. For large applications compiled with makefiles, Klocwork K7 and Coverity Prevent are the best choices. For small critical modules, PolySpace for C++ can do a very good job. Finally, for hybrid systems built with many languages and platforms, Parasoft Insure++ is recommended.

Sommaire

Practical verification & safeguard tools for C/C++

F. Michaud, R. Carbone; DRDC Valcartier TR 2006-735; RDDC Valcartier; novembre 2007.

Ce document est le rapport final d'un projet de recherche qui a été mené en 2005-2006. Le but de ce projet était avant tout d'identifier les défauts logiciels courants liés à l'utilisation des langages de programmation C et C++. Les erreurs et vulnérabilités créées par ces défauts ont aussi été étudiées, de manière à rendre possble la création de tests significatifs pour l'évaluation des meilleurs outils de vérification disponibles. Finalement, lorsque c'était pertinent, de bonnes pratiques ont été déduites de nos expériences avec ces outils. Puisque notre équipe avait déjà étudié les problèmes de design en 2002-2005, cette étude des problèmes d'implantation (ou de programmation) était logiquement la prochaine étape.

Nous avons constaté que C et C++ sont des langages très sujets aux défauts, erreurs et vulnérabilités logicielles en général, surtout pour les trois raisons suivantes. Premièrement, C et C++ donnent trop de contrôle au programmeur sur des aspects complexes et de bas niveau qui sont facilement mal utilisés, comme la gestion de la mémoire et l'arithmétique de pointeurs. Autrement dit, ils donnent suffisamment de corde au programmeur pour qu'il puisse se pendre. Ensuite, l'exécution de programmes C/C++ n'est pas supervisée par un moniteur et, à cause de cela, plusieurs erreurs comme les débordements ne sont pas détectés avant qu'il ne soit trop tard. Finalement, plusieurs aspects de ces langages ont été mal conçus (ex : tampons de taille fixe et chaînes de caractères terminées par un zéro) et ont tendance à exacerber les impacts d'autres problèmes.

Nous avons révisé plus de 30 outils séparés en cinq catégories et provenant de deux mondes : le monde universitaire/logiciel libre et le monde commercial/propriétaire. Nous avons constaté que les outils commerciaux étaient en général très supérieurs aux autres. Cela était prévisible, mais pas au point où la plupart des outils gratuits soient très limités (voire inutiles), alors que les commerciaux sont excellents. Par exemple, il existe plusieurs excellents outils de vérification pour les programmes Java. Nous croyons que c'est parce que l'analyse syntaxique des programmes C/C++ est très difficile à faire correctement, ce qui crée une barrière à l'entrée que seuls les projets bien financés (commerciaux) peuvent franchir.

De tous les outils que nous avons révisés, seulement cinq avaient le potentiel d'être vraiment utiles et ont été évalués plus en profondeur. Pour ce faire, deux ensembles de tests ont été créés : un ensemble de tests synthétiques et un autre avec du code en production. Les tests synthétiques étaient composés de 25 classes C++ contenant des défauts logiciels et intégrées dans une application de haute qualité dont le code source était libre. Les cinq outils ont tous bien accompli leur travail pendant ces tests : ensemble, ils ont détecté tous les problèmes sauf quatre. Les tests avec du code en production ont été faits à partir d'une petite application d'analyse numérique d'environ 10K lignes qui était un peu boguée. Cette fois, plusieurs outils ont eu beaucoup de difficultés à analyser l'application et n'ont permis de trouver à peu près rien. Il semble que du code de mauvaise qualité avec un style 'spaghetti' qui

utilise beaucoup les pointeurs peut être très problématique pour plusieurs outils. Toutefois, quelques outils ont été en mesure d'analyser l'application correctement et ont permis de découvrir plusieurs problèmes sans difficulté.

Notre principale recommandation est l'utilisation de meilleurs langages et plateformes, comme Java, C#, Ada ou tout autre langage de programmation moderne. Bien sûr, cela n'est pas toujours possible pour plusieurs raisons et l'utilisation d'outils de vérification peut être une avenue intéressante. Pour de grandes applications compilées à l'aide de /emphmakefiles, Klocwork K7 et Coverity Prevent sont les meilleurs choix. Pour la vérification de petits modules critiques, PolySpace pour C++ peut faire un excellent travail. Finalement, pour les systèmes hybrides bâtis à l'aide de plusieurs langages et plateformes, Parasoft Insure++ est recommandé.

Table of contents

Listings

List of figures

Introduction

Building a secure and reliable software system is a very difficult task. The degree of complexity involved is often substantial and one seemingly benign error can lead to serious consequences. As an example, *buffer overflows* are a well-known class of vulnerabilities caused by simple programming mistakes that can allow a remote attacker to execute arbitrary code on a vulnerable computer, essentially doing what he wants with it.

This document is the final report of an activity that took place in 2005-2006. The goal of this project was first to identify common software defects related to the use of the C and C++ programming languages. Errors and vulnerabilities created by these defects were also investigated, so that meaningful test cases could be created for the evaluation of best-of-breed automatic verification tools. Finally, whenever relevant, best practices were inferred from our experiments with these tools.

We focused on the level of software assurance generally needed for sensitive, but not safety-critical applications. Sensitive applications manage important and confidential information, hence they need to be at least a notch more reliable and secure than mass-market software, which have a poor record on these matters. However, these reliability and security requirements are nothing compared to what is necessary for safety-critical applications, which need a formal proof that some properties will always hold and that some things could never possibly happen.

We also focused on familiar technologies that people generally want, like the C and C++ programming languages and the Microsoft Windows and Linux platforms. There are other languages and platforms more robust than these (e.g. Ada and Java), but as of now, they are not used as much. However, we believe that, in the long term, people will get tired of being burned by security incidents caused by vulnerable technologies and will migrate to more robust ones.

Bugs in a software system come mostly from two sources: the design or the implementation. Our team previously worked on design issues through two activities: the Socle UML verifier (2002-05) and a command & control secure design patterns catalogue (2004-05). This verification tools study followed, focusing on implementation (or programming) flaws.

Following sensitive applications requirements, we were mostly interested by flaws that can have an impact on security. Software security mechanisms, like access control and encryption, are generally related to the design. However, we found that a sizeable majority of vulnerabilities had their origin in implementation problems.

This document focuses on flaws created while using the C and C++ programming languages. Its goal is to help software makers build secure and reliable programs by:

- introducing common programming defects related to C/C++, their effects on the program and the vulnerabilities they can create;

- helping them understand the potential pitfalls & design shortcomings of the C and C++ programming languages;
- presenting what kind of tools for software verification and safeguard of C/C++ program are available, what are their capabilities and effectiveness;
- presenting an evaluation of some of these tools.

Some Terminology

The terminology of information systems security is diverse and often ambiguous. What is the difference between an error and a failure? What about programming defects and vulnerabilities? An excellent discussion about computer security terminology can be found in [1]. The following is a short summary of this paper, giving some conventional definitions used in this document.

An *information system* is an entity that interacts with other systems, which can be hardware, software, and of course human users. These other systems are the *environment* of a given system. The *function* of a system is what the system is intended to do. This is described in the *functional specification* of the system. The *behavior* of a system is the implementation of the system's function and it is described by a sequence of states. These states can be *internal* (not visible to the environment) or *external* (visible to the environment). The sequence of external states, or what the user perceives of the system, is the service provided by the system to its environment.

An *error* is an event that occurs when the behavior of a system diverges from its functional specification. Since the behavior is a sequence of states, an error signify that at least one (or more) state(s) diverge from what is expected. Again, the divergence can be internal or external. However, an internal error will generally have an external effect at some point, but this is not always the case. Consider a program that overwrites unused memory: this is an internal error that will never have an external effect, but an error anyhow because this is not an expected behavior for the program. When an error prevents the system from offering correct service to its environment, a *failure* occurs.

The cause of an error is called a *fault*. A fault can be internal (inside the boundary of the system, e.g., a programming defect) or external (outside the boundary of the system, e.g., a malformed command sent to the system). It is important to understand the difference between an error and a fault: an error is related to the execution of the system; it is an event that happens in time. A fault, however, is something tangible that exists. Of course, both are closely related, since a fault in a program might cause an error in its execution. A fault is said to be *active* when it produces an error; otherwise it is *dormant*.

A *vulnerability* is an internal fault enabling an external fault to cause an error with important consequences. For example, a program that will crash (error) if sent a malformed command (external fault), because of a lack of user input validation (internal fault). Without the internal fault, the external fault could not cause an error and without the external

fault, the internal one would never be activated. An *exploit* is a small program, often a simple script, that will exercise a vulnerability and allow an attacker to gain some control over the system.

As an example, let us look at Figure 1, which shows a system composed of subsystem A and subsystem B. Both subsystems receive a state (function parameters) from their environment, do some computations which generate a sequence of internal states, and finally output a state back to the environment (function's return value).

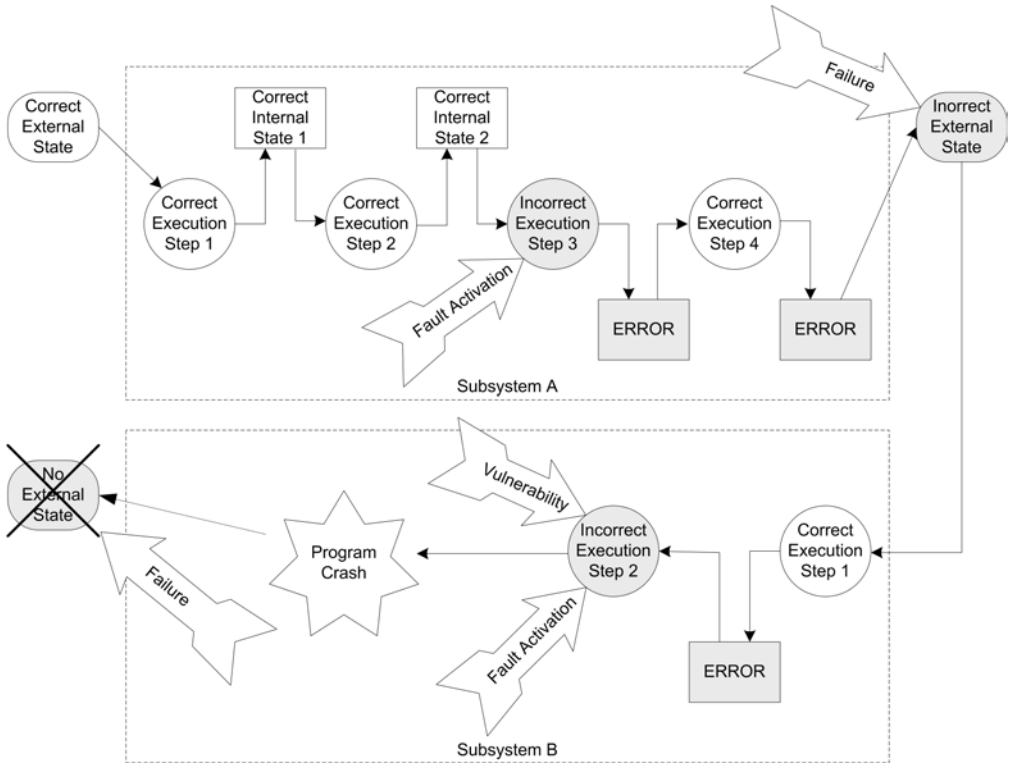

Figure 1: *The Relation Between Faults, Errors and Failures*

Subsystem A first receives a state from the environment (a character string) and do some computations on it. Everything is fine until execution step 3, which contains a programming defect (an index overrun in a string copy operation that will add 10 bytes to the null-terminated character string). The fault is activated by the computation and the next internal state contains an error (the character string is now bigger than expected). Even if the computation at step 4 is correct, since the input value is erroneous, the error is propagated (the string is still too long). As the string is returned, a failure occurs, because subsystem A is not offering a correct service to its environment: the string returned is longer than expected.

Subsystem B is fed with the string returned by subsystem A. Again, even if execution step 1 is correct, it does not prevent the error from being propagated; so the next internal state is erroneous. Then, another fault is activated at execution step 2: the 'too long' string is

copied into a local variable with strcpy, without first checking that the string can fit into the buffer. This creates another overflow, but this time, a pointer is overwritten with the NULL used to terminate the string. After that, the now NULL pointer is dereferenced, causing the program to crash. Execution step 2 contains a vulnerability, because an external fault (the too long string) activated an internal fault (use of strcpy without validation), which caused an error (buffer overflow) that had important consequences (program crash). Finally, the program crash prevents subsystem B from returning a value to the environment, which is by definition a failure.

Overview of the Task at Hand

The goal is to prevent the occurrence of errors, especially those that can lead to vulnerabilities. For this research and evaluation, we had no interest in the correctness of computations, since this would require knowledge about what the program is trying to do and a mechanism to verify if the actual behavior corresponds to what it should be. What is feasible, however, is to verify that the program executes 'normally', without doing anything incorrect for a C/C++ program. These incorrect behaviors are well known (see chapter 1) and given an execution trace, it is relatively easy to tell if a program has done something incorrect.

There are many ways to detect the possible occurrence of errors in programs. One can try to detect them directly, by executing the program and trying to detect states that correspond to an error. Instructions that can lead the program to an erroneous state are called program *defects*. Detecting defects is more difficult, because one has to guess in advance what will be the result of the execution of a program chunk.

There is also the question of what is an error and the answer is not the same for every verification tool. They do not try to find the same things because their definition of what is an error is not the same. For example, some tools will not consider overwriting unallocated memory an error if this memory is never used. In other words, some tools will only report defects when they have an effect on the program's execution.

C, C++ and Their Different Versions

In this document, C and C++ are often treated as if they were the same language, even if this is not the case. Most problems discussed here belong to C first, but since C++ is mostly a superset of C (a C program is generally a valid C++ program), both languages share a lot of problems. Besides, there are many versions of the C and C++ languages. This document refers to the ANSI C standard of 1989 and the ISO/IEC C++ standard of 1998.

Presentation of the Document

The first chapter presents vulnerabilities and errors that often emerge from the use of C/C++ programming languages, but that are not necessarily unique to them. Then, the following chapter shows how common programming defects can create these errors and

vulnerabilities, and why C/C++ are such a fertile ground for them. Chapter 3 presents a list of tools that can be used to detect program defects, limits the errors they can create, or reduce the chance of their occurrences in programs. The last chapter is a comparative evaluation of the best performing tools, followed by a conclusion.

This page intentionally left blank.

1 Vulnerabilities and Errors

Vulnerabilities allow an attacker to get some kind of control over an information system. The level of control gained can vary greatly, from being a nuisance to a disaster. Vulnerabilities are generally exploited in stages, where a seemingly benign error is used to create a series of errors that allow the attacker to build up the level of control he has over the system, finally reaching his goal.

The following presents three classes of vulnerabilities that can be the result of errors and defects in C/C++ programs. Of course, these vulnerabilities are not limited to C/C++ programs and can occur in similar environments.

Figure 2 shows the relations between defects, errors and vulnerabilities that are presented in this document.

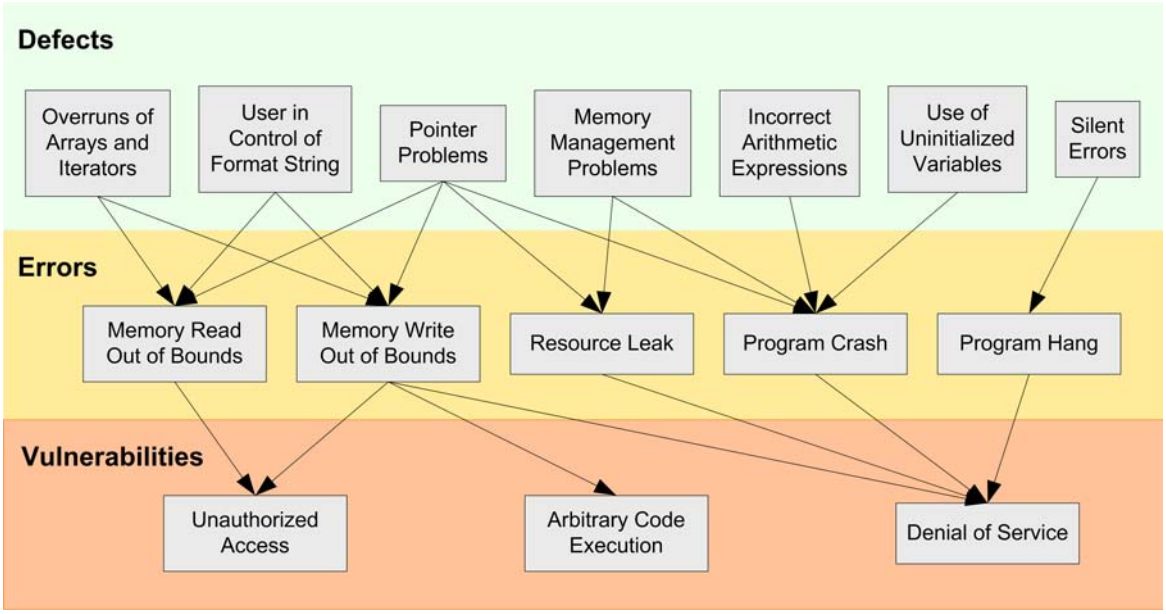

Figure 2: *The Relations Between Defects, Errors and Vulnerabilities*

1.1 Denial of Service

A denial of service vulnerability allows an attacker to prevent users from getting correct service from an information system. It can be done by stopping the service altogether or by slowing it down enough so it becomes unusable.

Stopping the service can be done by crashing the processes that support it. To do so, an unrecoverable error condition must be created, like the dereference of a null pointer or a division by zero. These conditions can be created with a memory out of bounds error, where a pointer or a variable is overwritten with a value that will cause the program to crash.

Defects such as overrun of local arrays and format strings in control of users can allow such conditions to be created.

A good way to slow down a system is to force it to use all of its available resources, such as memory. The lack of available physical memory will force the computer to swap memory to disk, which is a very slow operation. If a function in the vulnerable program is known to leak resources, an attacker can use the system in a way so that this function will be called repeatedly. This will cause the system to leak resources at a much more rapid rate than would be normally expected.

1.2 Unauthorized Access

An unauthorized access vulnerability allows an attacker to access part of a system without the required authorization. Since the control mechanism is usually implemented in software, it can be subject to manipulation. A software control mechanism generally looks like listing 1.

```
if (entered_password == expected_password)
    accessGranted = true;
else
    accessGranted = false;
return accessGranted;
```

Listing 1: *Simplistic Overview of an Access Control Mechanism*

An attacker with read or write access to memory could bypass the mechanism in at least three ways:

- he could read the expected password value in memory in advance and then enter it as a normal user would do;
- he could modify the value of accessGranted in memory so it is true;
- he could modify the program in memory (patch) to force it to always return true instead of accessGranted value.

All of these techniques require a read or write access to the program's memory and a knowledge of how the program is organized in memory. Defects such as buffer overruns and format string in control of users can allow such abilities.

1.3 Arbitrary Code Execution

This is the vulnerability with the most potential for a disaster. It allows an attacker to execute the code he chooses on the computer, essentially doing what he wants with it. The code to be executed can already reside on the computer, or it can be injected with the attack, which is more difficult but still very doable.

The trick is to take control of a vulnerable running program by redirecting its execution. This is usually done by overwriting the value of a function pointer that will be dereferenced in the future, so it points to the attacker's code. A good target is the return address of a calling function on the execution stack. Since local variables are stored on the stack too, next to the return address, an overrun of a local variable could allow an attacker to write to this location in memory.

A program can be vulnerable if an attacker is in control of data that will be copied in memory without validation for its size. As shown in Figure 3, the attacker exploit the fact that a character string copied in memory will overwrite what is next to it if it does not fit into the buffer.

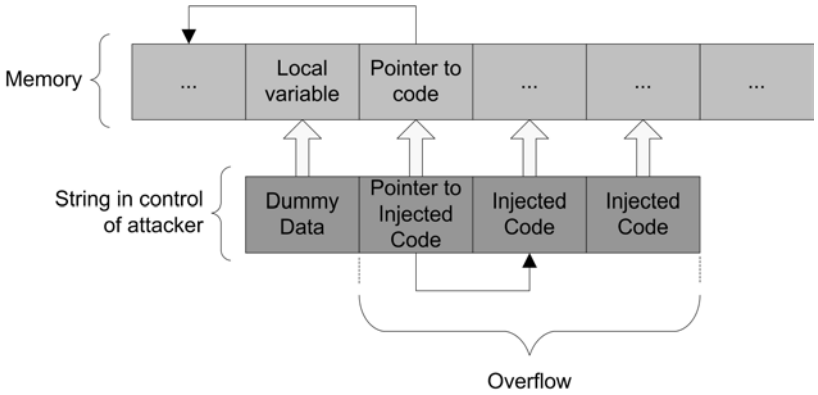

Figure 3: *A Generic Buffer Overflow with Code Injection*

The character string has to be cleverly designed to overwrite the pointer with the address of code to be executed. If there is enough space on the stack (a lot of space is allocated for local variables and the vulnerable variable is located as far as possible to the return address), it may also be possible to inject in memory the code to be executed.

1.4 Correct Program Execution

Many software security problems are caused by subtle manipulations of the execution environment of a program. To be able to understand these problems, one has first to understand how programs are executed, especially:

- how is a program organized in memory;
- how memory is allocated on the stack and on the heap;
- how functions are called and how they return.

The following is a short introduction to these key concepts. A more thorough discussion on these matters can be found in [2].

There are two important memory regions to consider when a program is executed: the execution *stack* (or just stack) and the *heap*. The stack is used to manage the flow of control and to pass parameters when functions are called. It also offers storage space to hold statically allocated variables (local variable in C/C++). The heap is the storage space for dynamically allocated (with malloc() and free()) variables.

1.4.1 The Execution Stack

The format of the execution stack is not the same for all computer architectures. Here, the conventions used by the Intel x86 architecture are shown (Figure 4).

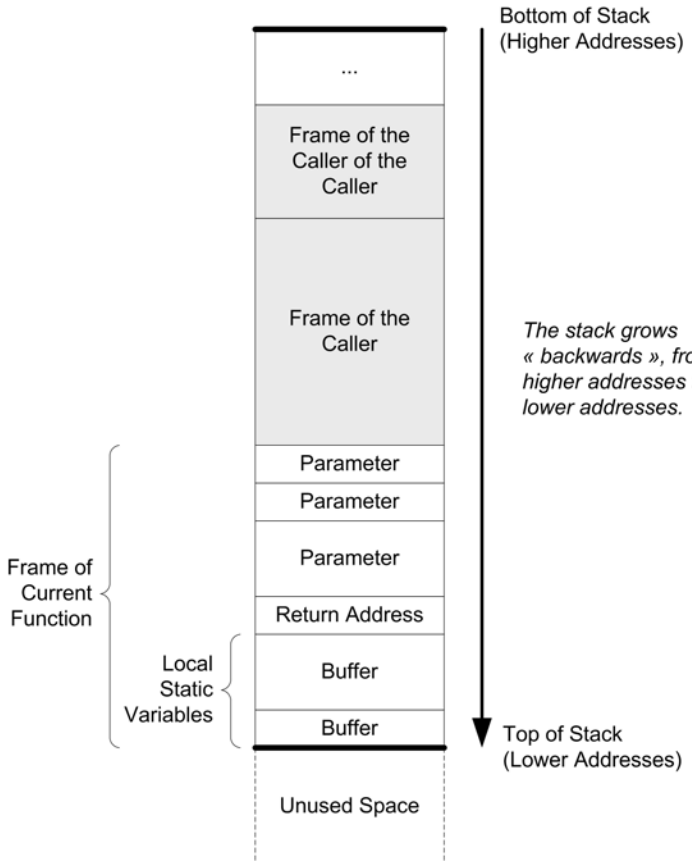

Figure 4: *The Execution Stack*

The stack is divided into frames. Each time a function is called, a new frame is pushed on the stack. On the x86 architecture, the stack grows 'backwards', from higher addresses to lower addresses.

When a function is called, what happens depends on the calling convention used. The C/C++ calling convention is as follows:

1. parameters to be passed to the called function are pushed on the stack;

2. the return address (the address of the next instruction following the function call) is pushed on the stack;
3. space for local variables is reserved on the stack;
4. control of the program is passed to the called function, which will read its parameters on the stack and use the reserved space for its local variables.

When a function returns:

1. the return value, if there is one, is stored in a special register on the processor;
2. the return address is read from the stack;
3. control of the program is passed to the calling function, by branching to the return address;
4. the calling function destroys the called function's frame by popping everything.

Accessing static variables on the stack with pointers should be done with caution, because nothing prevents an errant pointer to access memory outside the buffer's reserved space. Also, just next to local variables is the return address of the function, which could easily be overwritten by an overrun on the buffer sitting right next to it. If the return address is overwritten, when the function returns, the control flow will branch to this new and probably invalid address, generally crashing the program.

1.4.2 The Heap

The heap only contains blocks of dynamically allocated memory. Each block has a header (for management) and a storage space, as shown in Figure 5.

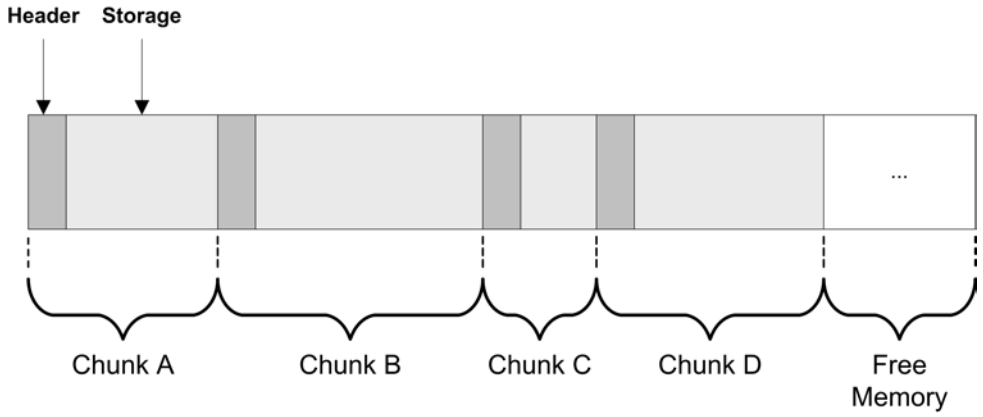

Figure 5: *The Heap*

The heap is a less fragile structure than the stack, because it does not systematically hold function pointers and its content is less foreseeable, because of relative unpredictability of dynamic memory allocation.

1.5 Errors

1.5.1 Memory Write Out of Bounds

This type of error occurs when a memory region is overwritten with invalid data, causing a condition generally known as a *buffer overflow* (see listing 2). The effect of an overflow depends on what is overwritten and with which value. A short introduction on the subject of buffer overflows can be found in [3]. For the curious reader, these two studies ([4, 5]) cover almost everything on the subject. Finally, a short tutorial on how to exploit buffer overflows can be found in [6].

```
void func(char *userdata) // userdata comes from the user {
    char buf[256];
    ...
    strcpy(buf, userdata); // userdata will overflow buf
    ...
}
```

Listing 2: *Write Overflow*

Overwrite of the Return Address on the Stack

The return address of a calling function is overwritten with a pointer to code that the attacker wants to execute. When the function returns, the execution is 'redirected'. This is one of the most common kind of buffer overflow, also known as *stack smashing* ([7]).

Overwrite of a Local Variable on the Stack

The value of a local variable, which always resides on the stack, is overwritten with another value. A clever attacker can overwrite specific values in a way to give him more control on a program. As an example, he could overwrite the variable containing is access level with a higher value, allowing him to access everything.

Overwrite of a Variable on the Heap

A dynamically allocated variable, which always resides on the heap, is overwritten. This situation is very similar to the overwriting of a local variable on the stack, but is harder to do. A good introduction to heap overflows can be found in [8].

Overwrite of a Part of the VTABLE (C++ only)

The VTABLE is used to store function pointers for classes. Each function pointer points to the code representing a class method. An overwriting of a value of the VTABLE can redirect the execution to arbitrary code when the method is called.

Not all write overflows can lead to vulnerabilities. A buffer overflow will be exploitable if it allows the attacker to:

1. inject in memory the code he wants to execute;

2. overwrite a pointer and redirect it to injected code;

3. force the execution to be redirected by the overwritten pointer.

1.5.2 Memory Read Out of Bounds

A read overflow is caused by the reading of a string that is not terminated by a null. The program then reads memory passed the buffer, until it encounters a null character. A read overflow will cause the program to read a string that may be a lot longer than planned, depending on where in memory is the next occurrence of a null character. The 'garbage' following the read buffer can affect the program in many ways, depending on what it does with the string.

Reading of Invalid Values

If the pointer does not point to a valid address, the value read while dereferencing it will also be invalid. This could lead to errors in computations that are not detected.

Reading of Sensitive Values

If the memory next to the read string contains a password or other sensible value, it can be read with the string and then printed on the screen or copied elsewhere.

Potential Domino Effect

A write overflow, overwriting the string termination null, can lead to a read overflow. Then, using a memory copy function without bounds checking to read this memory can lead to another write overflow, since the read data will be bigger than anticipated. This new write overflow can lead to another read overflow, and so on.

1.5.3 Resource Leak

A program consumes many resources while running: memory, disk space, network bandwidth, etc. These resources are available in finite quantities and should be returned to the 'available pool' when no longer used.

The allocation and freeing process generally works like this:

1. a call to an allocation function is made, with parameters defining options, like the size of the allocation;

2. if the allocation is successful (there is enough resource available), the allocation function returns a pointer to the allocated resource. This pointer is the only 'handle' the program has on the resource;

3. the resource is used by the program;

4. when the resource is no longer needed, the program calls a freeing function, passing it the pointer on the allocated resource;

5. the freeing function uses this pointer to identify the resource and frees it.

A resource leak occurs when the reference to the allocated resource (the pointer) is lost. When that happens, the program cannot free the no longer needed resource, because it has no way to tell the freeing function with resource it wants to free.

Many programs allocate and free resources in a loop. As long as there is as much resource freed than allocated, there is an equilibrium. However, when a program steadily consumes more resources than it releases, it is only a question of time before some resource become very scarce or is completely exhausted.

The impacts may vary from slowing down the system to crashing it, following the kind of resource that is leaking. A memory leak will force the system to slow down, because of memory paged to disk.

1.5.4 Program Crash

Some errors cannot be recovered, because there is no way to continue the execution of the program correctly. When these errors happen, the operating system stops the execution of the program immediately. Some of these irrecoverable errors are:

- dereference of an invalid (out of range or null) pointer;
- division by zero;
- hardware error;

If the pointer's invalid value is null or point to a memory address outside of the program's allocated memory, the program execution will be immediately aborted with a memory access violation (Windows) or segmentation fault (Linux) error message. The system will then become unavailable.

1.5.5 Program Hang

This kind of error is similar to a program crash, in that the system becomes unavailable too. However, the causes are not the same. When a program hangs, its execution is not stopped but instead is inside some kind of loop that never terminates, like a deadlock between threads.

2 Pitfalls of C/C++ and Related Defects

The C and C++ programming languages are powerful and useful languages. However, they should be used in specific cases where they are well-adapted to do the job.

C/C++ programs are fragile constructions without any monitoring or enforcement of rules. These constructions are a bit like a *Rube Goldberg* machine: they execute a task in stages, each stage counting on the correct execution of the precedent stage. These programs are easy to manipulate, because they execute blindly and a little change here and there can completely change their behavior.

If one wants to prevent errors in programs, one can do two things: first, build the program in a way to make sure that no error can happen. This is extremely difficult for non-trivial programs. Or, give the program the capacity to monitor itself and react to errors (e.g. make it *failfast*).

C and C++ are the worst of both worlds: their design allow the programmer to make errors that can radically change the behavior of the program and these errors will happen without any warning. Only the effects of the error will make it apparent that something bad happened and it could be to late.

The main reason why C and C++ are so fragile and not failfast is because of performance issues. Monitoring a program while it executes can be a heavy task. Since C is a low-level language with almost no execution overhead, it is best suited for high performance uses, like in operating system drivers, real-time applications, and in embedded systems.

This chapter presents some design shortcomings of the C and C++ programming languages that make it relatively easy for the programmer to create defects in programs.

2.1 Lack of Type Safety

A programming language is type-safe when it does not allow a program to treat a value as a type to which it does not belong. As an example, a variable that is supposed to contain a character string could not be assigned an integer. Also, since elements outside of an array (or buffer) have no type, accessing them is always a typing error. Hence, a type-safe language, by never allowing access to elements outside of a buffer, is immune to buffer overflows.

C and C++ are typed languages, but they are not type-safe. This is because their type systems are incomplete, allowing some constructions that are not type-safe. This is mostly because of the cast operator and the void pointer, which allow a typed value to be cast into another type, even if it is a mistake. A good example of this is listing 3: an integer (11111) is cast into a pointer and the pointer is then dereferenced, reading what is in memory at address 11111. This location is undefined and could contain any value. The variable may even be outside the range of the program's allocated memory.

```
...
char* c;
c = 11111;
printf("%c", *c);
...
```

Listing 3: *Example of Invalid Cast*

There are two kinds of type-safe languages: dynamically typed and statically typed. A dynamically typed language will do type checking at runtime. Then, if a typing error is detected, the program execution will be stopped. This can be a problem, because a program could contain typing errors and one will realize it only at execution time. Statically typed languages do their type checking at compilation time and give an assurance that typing errors cannot happen for all possible executions of the program. Many programming languages, like Java, are hybrids: they do as much type-checking as they can at compilation-time and then do the rest at runtime.

2.2 Pointer Arithmetic

Pointer arithmetic gives the programmer the possibility to alter the value of pointers arbitrarily, allowing him to read or write anywhere in the program's memory space. One legitimate use of this ability is to interface directly with hardware, using specific memory addresses as communication ports.

However, it is also sometimes used as a shortcut to access structures in memory in a more direct way, allowing performance optimizations.

The problem here is that it is too easy to make mistakes. Also, the effect of reading or writing 'somewhere' in memory can be unpredictable, making that kind of error very difficult to debug.

2.3 Buffers Have a Static Size

When a buffer has a static size, one has to foresee the largest possible size of the data that will be put into it. The programmer can pay a lot of attention to this necessity, but what about the user? If entered data is not validated for its size, it could be possible to enter data that does not fit into the buffer and create a *buffer overflow*.

To make things worse, if the oversized data is received from the network and the buffer vulnerable to overflow, that could mean a remotely exploitable vulnerability allowing the complete takeover of the computer. This is a good example of a small programming defect having potentially important consequences.

There are libraries available for dynamically allocated buffers, which adjust their size in function of the input size, but they are seldom used because of the overhead in program-

ming and performance degradation. Local buffers allocated on the stack are very efficient performance-wise.

2.4 Lack of String Type

The C language has no character string type. Because of this, strings are represented using arrays of characters (buffers), which have a static size. Being static buffers, C 'strings' can create the problems described in Section 2.3.

Since strings are used in almost every program, one should not have to take special care about their use. Also, an erroneous use should not have the potential to create serious vulnerabilities, which is the case with C/C++.

The C++ language has a string type in the Standard Template Library (STL). However, many C++ programmers still use the C way of managing strings and do not take advantage of the STL string class.

2.5 Null-Terminated Strings

Programming languages use different data structures to represent strings in memory. Pascal, for example, reserves a byte at the start of the string that tells its length. This means that the maximum length that a string can take in Pascal is 255 bytes. C, on the other hand, put a special character (zero or NULL) at the end of the string to tell where the string ends. This allows for arbitrary length strings that are limited by available memory only. However, one has to traverse the string until a null character is found to know its length. Then, if the string-termination NULL is overwritten or if the programmer forgets to add it, it becomes impossible to know where the string ends. A program reading a string would then continue to read memory (a *read overflow*) until it encounters another NULL, leading to many problems.

Besides, when looping on a C string, the programmer has to check for two stopping conditions:

1. the NULL character indicating the end of the string;
2. the index used to traverse the string should be inside the bounds of the array.

The second condition is a safeguard in case the null character was replaced by something else, because of a write overflow that happened earlier, or just because the programmer forgot to end the string with a NULL. This second condition is often ignored, but should not.

2.6 Standard Libraries Containing Hazardous Functions

Many string manipulation functions do not check the bounds of the destination buffer to ensure that it is not overwritten. If the programmer is not aware of this situation, it could

lead to dangerous problems like buffer overflows.

Here is a non exhaustive list of those risky to use functions: strcpy(), strcat(), sprintf(), and gets(). These functions should be replaced with cousin functions, like strncpy(), strncat(), snprintf(), and fgets(). These functions have an additional parameter that limits the number of bytes copied. If the size of the destination buffer is passed to this parameter, an overflow cannot occur.

The scanf() family (scanf(), fscanf(), sscanf(), vscanf(), vsscanf(), and vfscanf()) also do not check the bounds of the destination buffer and they have no replacement available. When these functions are used, the length of the source string should be checked, since they do not control the size of the destination buffer.

2.7 C99: A Better C

The 1999 version of C brought many changes to the language, generally to ease the programmer's job and standardize some language extensions that were already supported by many compilers. However, these changes have few effects on safety and security, except some that can help to prevent buffer overflows.

2.7.1 Variable Length Arrays

C99 supports arrays whose size is known only at runtime. However, once the array is allocated, its size cannot change.

```
...
void vlaFunction(int n)
{
    // During the lifetime of a vla, its size cannot change
    int vla[n];
    // sizeof evaluated at run-time, not at compile-time
    printf("%d\n", sizeof(vla)); // sizeof(int) * n
}
...
```

Listing 4: *Variable Length Arrays*

This does not prevent a buffer overflow, since the array will not grow to accommodate the allocated data if it is bigger than the array's size. However, this opens the door to a correct and easy management of array size, where an array is allocated at just the right size when needed, instead of having a static maximum size giving enough leeway for all expected situations and often wasting memory. Nevertheless, since the array size must be managed manually, there is still a risk of making a mistake.

2.7.2 Standard Library Changes

Many functions vulnerable to buffer overflows, as sprintf, have been replaced by correct ones, like snprintf. These functions existed before, mostly in BSD and Linux libraries, but

were not 'officially' part of the Standard C Library. With C99, they are. However, many old and vulnerable functions, like gets, were not removed from the library.

2.7.3 C99 Compiler Support

Complete support for C99 is hard to find, especially with mainstream compilers. For example, Microsoft and Borland compilers do not offer support and will not in the short term. However, many less known compilers, like GCC and Intel's, support the standard partially, including some of the most important new functionalities, like variable length arrays and standard library changes.

2.8 On Defects

Most defects are not 'always on', meaning that they will not generate errors every time they are executed. They are program constructs that *could* lead to execution errors. The activation of the defect often depends on many conditions (e.g. the size of the data entered by an user) that should be met for the error to happen.

Furthermore, most defects are composite and cannot be attributed to only one single program instruction. It is a sequence of instructions that constitutes a defect and individually, all the instructions in the sequence could be correct. This means that any serious attempt at detecting defects requires a deep understanding of what the program is doing. A simple syntactic search for 'bad' instructions in the source code will not work very well. To complicate matters, the defect can also be in the absence of something that should be there, like data validation, error management or unused resource release.

This section presents well known programming defects generally associated with the use of the C and C++ programming languages. The following list of defects is built as follow: first, a short description of the defect is shown, with a code example if necessary. Then, the common causes of the defect are explained. Finally, errors caused by the defect are shown, with associated vulnerabilities. The reader can find more details in this excellent guide on secure programming: [9].

2.9 Memory Management Problems

This section presents defects associated with the management of memory.

2.9.1 Use of Freed Memory

The use of freed memory is dangerous, because the operating system can 'recycle' memory that was freed at any time. Reused memory will be modified, overwriting the values that were expected by the program and causing unpredictable effects.

```
char* p;
p = (char*) malloc(sizeof(char));
*p = 'a';
...
free(p); //memory pointed by p is freed before being used
...      //new memory is allocated, overwriting the old one
printf("%c", *p); //the value of *p is invalid
```

Listing 5: *Use of Freed Memory*

2.9.2 Underallocated Memory for a Given Type

When allocating memory for a structure with malloc(), the programmer must know the right size for the allocation. The **sizeof** operator can help remove the guesswork, but mistakes are still possible.

```
struct S {
    int a;
    int b;
    char c[100];
};

struct S* f1(void) {
    struct S *p;
    p = (S*)malloc(sizeof(p)); //Allocation too small
    return p;                  //sizeof(*p) was intended
};
```

Listing 6: *Underallocated Memory for a Given Type*

If not enough memory is allocated, some will be overwritten with invalid values when the structure is assigned data. This will have unpredictable effects.

2.9.3 Freeing of Unallocated Memory

Passing an invalid pointer to free() can have unpredictable results. On most platforms, the implementation of the free() function will cause the program to crash with a memory access error if an invalid pointer is passed to it.

2.9.4 Incorrect Array Deletion

This problem is specific to C++. Calling delete instead of delete [] on a dynamically allocated array is not guaranteed to always work. Moreover, if the array's elements have destructors, they will not be called.

```
char *buf = new char [10];
...
delete buf; //ERROR, should be delete[] buf.
```

Listing 7: *Incorrect Array Deletion*

2.9.5 Overlapping Pointers in memcpy()

The pointers passed to memcpy() must point to non-overlapping memory regions, otherwise the results are undefined and may vary following the implementation.

2.10 Overrun of Arrays and Iterators

This section presents a class of defects that often lead to out of bounds memory read or write errors.

2.10.1 Overrun of a Statically Allocated Array

```
int buf[10];          //buff has 10 elements, index 0 to 9
int *x = &buf[1];     //x now points to buff[1]
x[9] = 0;             //x[9] is equivalent to buff[10],
                      //which is out of bounds
```

Listing 8: *Overrun of a Statically Allocated Array*

2.10.2 Overrun of a Dynamically Allocated Array

```
// allocates 40 bytes (10 * 4 byte ints)
int *buffer = (int *) malloc(10 * sizeof(int));
// valid indices are buffer[0] to buffer[9]
int i;
// overruns memory by writing buffer[10]
for (i = 0; i <= 10; i++) {
        buffer[i] = i;
}
```

Listing 9: *Overrun of a Dynamically Allocated Array*

2.10.3 Overrun of an Iterator

```
list<int> *li = new list<int>();
list<int>::iterator i = li->end();
int x = *i; // ERROR: dereferencing past-the-end
```

Listing 10: *Overrun of an Iterator*

2.11 Pointer Problems

2.11.1 Returning a Pointer to a Local variable

A pointer to a local variable becomes invalid as the execution of the program exits the scope of the function. In C/C++, local variables are stored on the stack and they can be overwritten by the call to another function once they get out of scope.

As an example (listing 11), a call to a function between a call to getValue() and the use of return pointer p will overwrite the variable pointed by the pointer.

```
main(){
    char* p;
    p = getValue();
    ... // call to another function with local variables
    printf("%c", *p);
}

char* getValue(){
    char c;
    c = 'x';
    return &c; // the address of c is returned just
               // before it becomes out of scope
}
```

Listing 11: *Returning a Pointer to a Local variable*

Of course, the address of a local variable should never be returned by a function. Instead, one should dynamically allocate the variable if it is to be used outside of the function.

2.11.2 Incorrect Pointer Arithmetic

A direct access to a data structure using pointer arithmetic is not done correctly. Even if it is legal to do so in C/C++, the programmer should let the compiler calculate memory addresses to access structure elements by using the dot notation, thus reducing the chances of errors.

2.11.3 Dereference of a Null Pointer

Dereferencing a null pointer will cause the program to crash.

2.11.4 Unchecked Null Return

Many allocation function (for memory, file, socket, etc.) will return a pointer on the allocated resource, or null if the allocation was unsuccessful. The returned pointer should always be checked for a null value before being dereferenced.

```
struct date
{
    int year;
    int month;
    int day;
}
...
struct date d;
int *p; // pointer for direct access of struct elements
p = &d;
d.year = 2004;
d.month = 11;
d.day = 22;
...
// there is a typo in the direct access of the day
printf("The date is: %d %d %d", *p, *(p+1), *(p+22));
// date.year, date.month and date.day should be used instead
```

Listing 12: *Incorrect Pointer Arithmetic*

```
void f(int* p){
    int x;

    if (p == NULL)
        x = 0;
    else
        x = *p;
    *p = x;    // ERROR: p is potentially NULL
}
```

Listing 13: *Dereference of a Null Pointer*

```
struct S
{
    int a;
    int b;
    char c[100];
};

int main(void)
{
    struct S *p;
    p = (struct S*)malloc(sizeof(struct S));
    p->a = 0; // ERROR: defererencing p without first checking
              // if it is null.
    return 0;
}
```

Listing 14: *Unchecked Null Return*

2.11.5 Bad Pointer Cast

A pointer is cast to a wrong type, leading to errors in the interpretation of memory values. As an example, listing 15 shows an int that is misinterpreted as a short.

```
...
int a, b;
void *p;
p = &a;
a = 100000;
b = *(short*)p;
...
printf("%d %d", a, b); // Will print: 100000 -31072
...
```

Listing 15: *Unsafe Pointer Cast*

Void pointers are often used as polymorphic data types and this practice can lead to typing errors that will not be catched by the compiler. In listing 16, an error in the value of 'type' could lead to a bad cast.

```
// x is a pointer to a value of many possible types.
int polymorphicFunction(void *x, char type)
{
    ...
    switch{type}
    {
        case 'i': ... // interpret the value of *x as an int.
        break;
        case 's': ... // interpret the value of *x as a short.
        break;
        case 'd': ... // interpret the value of *x as a double float.
    }
    ...
}
```

Listing 16: *Use of Void Pointers as Polymorphic Data Types*

While it can be practical and efficient to use void pointers as polymorphic data types, one should be careful to use them correctly.

2.11.6 Resource Reference Lost

This type of defect leads to resource leak errors.

A pointer on allocated memory was reused before freeing the allocated memory.

Reusing variables is considered a bad practice, except of course inside a loop where a new variable cannot be declared at each iteration. Reusing variables makes the program less understandable and increases the chances of the programmer making a mistake.

```
...
char* p;
p = (char*) malloc(sizeof(char));
...
p = (char*) malloc(sizeof(char)); //p was not freed before
...                                //being reallocated
```

Listing 17: *Pointer Reassignment Without Prior Deallocation*

2.12 User in Control of Format String

Functions using format strings, like printf() or scanf(), can lead to dangerous problems generally known as *format string vulnerabilities*. A good overview of the problem is presented in [10] and [4]. Details on how to exploit this kind of vulnerability can be found in [11], [12], and [13].

The problem occurs when the user can get in control of the format string. This can happen if a user input is directly passed to the function, without separate format string parameters. Listing 18 shows a dangerous use and listing 19 a safe use.

```
int printUserName() {
    static char input[100] = {0};
    printf("Enter your name: ");
    fgets(input, sizeof(input), stdin);
    printf(input); // This is where the problem lies.
    return 0;
}
```

Listing 18: *Dangerous Use of a Function Using Format Strings*

```
int printUserName() {
    static char input[100] = {0};
    printf("Enter your name: ");
    fgets(input, sizeof(input), stdin);
    printf("\%s", input); // No problem here.
    return 0;
}
```

Listing 19: *Safe Use of a Function Using Format Strings*

The main cause of this kind of defect is ignorance of the problem. Many programmers are simply unaware of format string vulnerabilities. Since a safe use of format string functions is not restricting in any way, there is no reason to use them incorrectly.

A dangerous use of a format string function will not cause an error in itself. However, if a malicious user can get in control of the format string, he will probably try to create a write overflow in a way to take control of the vulnerable process, as shown in [12].

2.13 Silent Errors

2.13.1 Ignored Return Value

This is not necessary a flaw. However, many functions use return value to signal an error condition and these should always be checked.

Errors that go unreported by the program can affect the execution in many ways, like giving an erroneous result or telling the user that a given task was done successfully when this is not the case.

2.13.2 Empty Catch Statement

This defect, specific to C++, is similar to an ignored return value in its effect: it can create a silent error.

```
...
try
{
    ... // statement that generates an exception
}
catch (ExceptionType e)
{
   // catch the exception and do nothing
}
```

Listing 20: *Empty Catch Statement*

There is no common cause for this kind of defect. Often, when writing code, the programmer leaves the catch statement empty at first, to be completed later. This is because error management logic is not always specified in the design and can be hard to do well. Then, the programmer forgets to complete it and the final code contains this defect.

2.14 Incorrect Arithmetic Expressions

Some arithmetic operations on numbers can lead to errors on computations and program crashes.

2.14.1 Division by Zero

A division by zero will generally be handled directly by the processor, which will trigger an interrupt to be catched by the operating system. The operating system will then stop the execution of the problematic program.

2.14.2 Integer Overflow

Data types used to represent numbers have a maximum size following the number of bits used.

```
unsigned char a = 255;
unsigned char b;

b = a + 1; // b will be equal to 0, not 256
printf("%d", b);
```

Listing 21: *Integer Overflow*

2.15 Use of Uninitialized Variables

C and C++ do not automatically initialize local and class variables. Uninitialized variables contain unpredictable values. In this case, the pointer was never initialized and points to an unpredictable memory location.

```
...
char* p;           // p is declared but never initialized
printf("%c", *p); // the value of p is undefined when used
...
```

Listing 22: *Example of Uninitialized Pointer*

Most compilers can detect simple instances of this kind of defect and warn the programmer. However, complex allocation schemes involving nested structures will not be detected.

3 Tools Overview

This chapter presents an overview of the tools that can detect programming defects, limit the errors they can cause or reduce the chances of their occurrence in code.

All tools in this chapter were not formally evaluated, but we worked with them long enough to have a good idea of their potential. Those that had a very good potential were then formally evaluated (see next chapter).

We identified five categories of tools, each with their advantages and inconveniences:

- Library Replacements
- Program Hardeners
- Static Analyzers Based on Program Syntax
- Static Analyzers Based on Program Semantics
- Runtime Testers Using Code Instrumentation

The tools evaluated come mainly from two worlds: academic/open-source and proprietary/-commercial. Academic and open-source tools are often the result of research projects experimentations or are quick hacks made by a programmer on his free time. Because of this, many of them are rudimentary and very focused, come with incomplete or no documentation and are in general of limited use. Of course, there are notable exceptions to this rule. Nevertheless, almost all of them are free and they are a good start for someone who wants to be introduced to software verification without investing a lot of money. On the other hand, commercial tools are generally well made and very helpful. Most of the time, they can detect hard to find bugs in a completely automated way. However, almost all of them are very expensive.

The list of tools in this document is not exhaustive. New free and commercial tools appear on the market every month and it is difficult to keep track of all of them. Also, a demonstration version of expensive commercial tools is not always available. Because of this, some high-end commercial tools were not evaluated, but they are nonetheless discussed.

As the reader will see, an holistic approach using many kinds of tools is preferable, because no tool is a silver bullet. All tools we have seen can currently only detect low-level problems, like buffer overflows and memory leaks. Higher level design problems, like access control and security in general, are still out of reach.

3.1 Library Replacements

These libraries are used to replace dangerous to use string manipulation functions (see Section 2.6) found in standard libraries. They offer functions with similar functionalities, but without the vulnerabilities. Some libraries require no modification to the source code, since

the interfaces and functionalities are exactly the same. Other libraries offer more advanced functionalities, but these cannot use the old interfaces and thus require modifications to the source code.

3.1.1 Better String Library

Description: The bstring library is an attempt to provide improved string processing functionality to the C and C++ languages. At the heart of the bstring library is the management of 'bstrings', which are a significant improvement over null terminated char buffers (see Section 2.5).

Buzz: This is an excellent initiative which provides not only an enhanced string library for C and C++ which is more secure than the standard libraries, but it can also be used by different compilers on different platforms. While there are commercial alternatives available, they rarely are portable beyond one or two platforms.

Status: This project is still under active development and it is currently at version RC15. It was developed by Paul Hsieh. The project appears to have been started in December 2002.

Web: http://sourceforge.net/projects/bstring/

Licence: BSD.

3.1.2 LibSafe

Description: LibSafe is based on a software layer that intercepts all function calls made to library functions known to be vulnerable. A substitute version of the corresponding function implements the original function in a way that ensures that any buffer overflows are contained within the current stack frame, which prevents attackers from overwriting the return address and hijacking the control flow of a running program.

Buzz: The true benefit of using LibSafe is protection against future attacks on programs not yet known to be vulnerable. The performance overhead of LibSafe is negligible, it does not require changes to the OS, it works with existing binary programs, and it does not need access to the source code of defective programs, or recompilation or off-line processing of binaries. LibSafe is supposed to be platform independent and it has apparently been ported to other architectures such as Solaris and BSD. However, it is no longer possible to find the source code easily. The only versions of LibSafe that are readily accessible are the precompiled version available from rpmfind.net. Source code may be included with certain Linux distributions, but then the code may have to be changed to fit a particular architecture and platform.

Status: It does not appear as though this project is still under active development. It cannot be determined when the project was started or who the original developers were. The last developers to work on the project were Navjot Singh and Tim Tsai. LibSafe is

a library that was originally developed by Lucent Technologies (formerly Bell Labs). The current version is 2.0.

Web: http://www.gnu.org/directory/libsafe.html

Licence: GNU GPL.

3.2 Program Hardeners

Program hardeners make modifications to a program or to its execution environment in ways to reduce the effects of defects and vulnerabilities that could be present in the code. This generally slows down the execution a little and may require a special configuration. However, program hardeners are a good way to protect a system from future attacks.

3.2.1 Determina Memory Firewall

Description: The memory firewall enforces security through the use of several fundamental checks of software conventions, enforced in thousands of locations within a typical software application. These locations are determined automatically by inspecting the software code the first time it is run. The two primary categories of checks are the following:

- **Enforce Valid Code Origin** – Security checks are performed on the origin of code to ensure that it is valid. This prevents unintended injected code from being executed. It also catches malicious code masquerading as user data.
- **Enforce Valid Control Transfer** – Security checks are performed on the program control flow to ensure that control transfer always conforms to software conventions. This prevents applications from being tricked into handing over control to external injected code.

Buzz: The technology behind this product, called *program shepherding*, is impressive and truly innovative. The curious reader can get an idea on how it works by reading [14]. The memory firewall does not prevent buffer overflows from occurring – an attacker can still use the vulnerability to overwrite memory. However, the execution of the vulnerable process can no more be redirected to the injected code. In other words, the memory firewall make buffer overflow vulnerabilities a lot less dangerous.

Status: This is a new product that went on the market at the end of 2004.

Web: http://www.determina.com/product/overview.asp

Licence: Commercial.

3.2.2 Stack Shield

Description: Stack Shield is a development tool that protects against 'stack smashing' attacks (see section 1.5.1) without requiring any change in the code.

When a function is called, the Stack Shield protection system copies the return address in a location where overflows cannot occur and checks if the actual and copied values are different before the function returns. If the two values are different, the return address has been modified so Stack Shield terminates the program or try to let the program run ignoring the attack, risking at maximum a program crash. Stack Shield is also able to stop frame pointer and function pointer overwrite attacks.

Buzz: Stack Shield works as an assembler file processor and is supported by GCC/G++ front ends to automatize the compilation. No code change or other special operations are required (only makefiles need minor changes). This is an excellent program that was developed for use on the i386 architecture and under Linux, but it will also work under Cygwin on Windows. However, it should be possible to port it to other platforms and architectures. The protection offered by Stack Shield is not 100% effective and there are ways to bypass it, as seen in [15].

Status: This project is no longer under active development. It was developed by Vendicator. This project has not been updated since 2000. It is currently at version 0.7. The first public release was in 1999.

Web: http://www.angelfire.com/sk/stackshield/index.html

Licence: GNU GPL.

3.2.3 DieHard

Description: DieHard is an application memory allocator replacement that changes the way memory is allocated on the heap. It can protect against heap memory errors, such as buffer overflows, dangling pointers, and invalid frees. It works by randomizing the location of objects in a heap that is larger than required, leaving unused memory between these objects. Then, when memory is written out of bounds, chances are that the memory overwritten will be the unused 'space' between objects. Because of the randomized location of objects, exploiting a vulnerable overflow to inject code becomes much more difficult, if not impossible, to do reliably. DieHard can also run many instances of a program simultaneously (replicated mode), where the randomization of the location of objects is different for each replica, thus reducing the probability that a memory error will affect all replicas simultaneously. When an error occurs, the execution continues with an unaffected replica. More details on this project can be found in the following document: [16].

Buzz: Using DieHard will increase memory consumption by 50-75%, or even more if using the replicated mode, but it will not affect the execution speed significantly. DieHard works with almost any application on Linux, but on Windows it only offers protection for the Firefox web browser for the moment. The replicated mode also only works on Linux. While being a very good idea, DieHard only protects from memory errors on the heap (it does nothing for the stack) and the protection it offers is only probabilistic and is no real assurance.

Status: This project is in active development, currently at version 1.0.1.

Web: http://www.diehard-software.org

Licence: Free for non-commercial use.

3.3 Static Analyzers Based on Program Syntax

These tools do not really try to understand the semantics of the code, a task that can be very difficult with C/C++. Using regular expressions and common parsing techniques, they do a simpler syntax analysis of the source code to find suspicious constructs. Because of this superficial analysis, they have a tendency to generate many false positives and can only detect simple problems. However, their simpler analysis also makes them more robust at handling code that contains extensions to the language, because these extensions are simply ignored (but there are exceptions). Also, since syntax analysis is not a computation-intensive task, they are generally very fast and can handle programs of any size.

An evaluation of most of the tools of this section can also be found in the following papers: [17], [18], [19].

3.3.1 Secure Programming Lint (Splint)

Description: Splint is a static analysis tool for C source code. It can find problems with dereferenced null pointers, incorrectly used or implemented storage types, memory errors including overflows and underflows, as well as memory leaks. It can also detect many other kinds of problems, such as infinite loops. It compiles under most platforms and strictly supports the ANSI C programming standard. For the reader who would like a quick start, a good paper on Splint use is available [20].

Buzz: In many ways, it is a powerful and versatile tool. Unfortunately, because of its strict support for ANSI C, it cannot parse programs that were written with extensions to the language, a very common practice. Thus, the use of Splint often requires modifications to the source code and this can be very impractical for large programs. Also, the advanced features require the use of annotations in the source code. This allows Splint to use very clever analysis techniques, but is again impractical for large existing source code without annotations. Finally, like other static analyzers based on syntax, it tend to generate many false positives.

Status: It is difficult to determine when this project actually started. The project has been registered with SourceForge since August 2000. It is currently at version 3.1.1. It appears to still be under active development. The authors of the program are David Larochelle, David Evans, Hebert Martin Dietze, Mike Lanouette, and Mark D. Babushkas. The project is funded by NASA Langley, NSF, and Usenix.

Web: http://splint.org/

Licence: GNU GPL.

3.3.2 PScan

Description: PScan is another static analysis tool that is a limited problem finder. While it is not as powerful as Flawfinder and Splint, it still serves a purpose, since it is particularly well suited for finding format strings problems.

Buzz: PScan is a good little tool that does not do much, but does it quickly and accurately. It is possible to expand upon the information it reports by specifying the flags '-v' and '-w', but not in conjunction with the other. PScan does not generate many false positives. This is a tool for novices and experts alike, but if the need is a fully fledged bug finding tool, the use Splint or Flawfinder would be a better choice than this command-line based tool.

Status: This project is no longer under active development. It was last developed in July 2000 by Alan DeKok. It cannot be determined when the project was started. It also cannot be determined what version the project is currently at.

Web: http://seclab.cs.ucdavis.edu/projects/testing/tools/pscan.html

Licence: GNU GPL.

3.3.3 Flawfinder

Description: Flawfinder is a powerful static analysis tool that supports both C and C++. It is a Python-based program that contains a built-in database of well-known security holes in C and C++ code, such as buffer overflows, string formats, race conditions, etc. Potential security risks are sorted by their possible security weakness on a scale of 1 to 5, where five is the most dangerous and 1 is a small problem.

Buzz: Luckily, Python exists on most platforms, so it can run on almost anything. While Flawfinder will generate false positives from time to time, it tends to be quite accurate. The project is based on the assumption that it is better to generate some false alarms rather than miss potential security holes. A powerful feature that Flawfinder supports is that it scans entire directories at the same time, minimizing the analyst's time for specifying files. Therefore, if one has large projects with many hundreds of source code files, one can use Flawfinder in batch mode. Flawfinder seems to be widely used by many open-source projects on the Internet (e.g. the OpenBSD group uses it to audit their source code). Flawfinder is a command-line based tool, but with an optional switch it can convert all results into HTML for review with a web browser.

Status: The initial version 0.12 was released in May 2001. It is written by David A. Wheeler. The project is currently at version 1.26 and it is still under active development.

Web: http://www.dwheeler.com/flawfinder/

Licence: GNU GPL.

3.3.4 Rough Auditing Tool for Security (RATS)

Description: RATS is a source code analyzer that checks for common security errors introduced through programming, including race conditions. It can work with multiple languages, which include C, C++, Perl, PHP, and Python.

Buzz: Very few tools in this category check for race conditions or can work with multiple languages. These capabilities make RATS a versatile tool. It is generally very fast and can accept directory names as one of its arguments, allowing many source code files to be analyzed in batch mode. In one of our experiments, it was able to process 57765 lines of C and Perl source code in less than 1 second and its output, during post-analysis, was relatively accurate when compared with other tools such as Flawfinder and Splint. It should compile on just about any major platform.

Status: It cannot be determined when the project RATS was started, nor can it be determined who actually developed RATS. It has been developed by Secure Software that was incorporated in 2002. They have made their RATS program freely available via the GNU GPL license. RATS is currently at version 2.1.

Web: http://www.securesw.com/download_rats.htm

Licence: GNU GPL.

3.3.5 ITS4

Description: ITS4 is another static analysis tool that is very similar to Flawfinder and Splint. It is very fast and provides easy-to-understand messages without too much technical details. It works equally well on C and C++. The interested reader can find a good description of ITS4 in the following paper: [21].

Buzz: Unlike most of the tools in this category, this easy-to-use program has a good support for C++. Unfortunately, this command-line based tool cannot verify all source code files in a directory and one has to specify either a file name or a file type.

Status: This is not an open source project but rather it is a commercial tool. However, the company that created it, Cigital, makes it available for free. It appears as if the first release was version 1.0.1. It is currently at version 1.1.1. Because the source code does not appear to have been changed since 2001, this program does not seem to be actively maintained.

Web: http://www.cigital.com/services/its4/download.php

Licence: Proprietary commercial license.

3.3.6 Smatch

Description: Smatch is a C source code checker that was designed to analyze the Linux kernel's source code. Smatch is itself a patch that must be applied to the appropriate

version of GCC and then be recompiled for use. This modified GCC is then able to create at compile time '.c.sm' files that are piped out to specialized scripts that the individual programmer must write himself in order to find errors in his source code. The .c.sm files contain a modified form of the source code's information that can be more easily analyzed through the use of scripts.

Buzz: Smatch is a good attempt at a personalized static analysis tool. However, if one does not know what kind of problems to look for and how to detect them, then this tool is of no use. In other words, this is a tool for experts only and it could be quite useful as a jump-start to create a home-made static analyzer. A more recent update would be needed by Smatch because it currently only works with GCC 3.1. This project should still be considered as being in its alpha stage. In addition, it is a command line based tool.

Status: The project still appears to be under active development. It is currently at version 0.50. It was developed by Dan Carpenter. The project appears to date back to June 2002 when version 0.03 became public.

Web: http://smatch.sourceforge.net/

Licence: GNU GPL.

3.4 Static Analyzers Based on Program Semantics

The is one of the most interesting category of tools we found, because of their effectiveness, versatility and extensibility. These static analyzers are generally composed of two modules: a compiler front-end and an analyzer back-end. The front-end will parse the source code and generate a model of it, on which the analyzer back-end will work to find problems and bugs. Many tools support the customization of checkers used by the analyzer, allowing it to detect new ad hoc problems.

To generate a model of the program, the front-end requires that the program compiles without error, allowing the verification of partial (non-completed) programs during development. However, with C/C++, this is not as easy as it seems. First, many compilers allow the use of non-standard extensions to the language and these are used often. However, many tools front-end are strict and will not accept a C/C++ program that contains these extensions. Because of this, it is often necessary to modify the non-standard source code before being able to analyze it, which can be impractical for large programs.

A related problem is the common use of big, complex and monolithic makefiles. These scripts drive the compilation process and use an often very complex configuration, where some parameters come from environment variables, other from configuration files, etc. We found that, for large programs compiled with makefiles (e.g. the Firefox web browser), the only way to make it work without errors is to reinstall an entire computer's operating system, with many small utility programs, and to configure every detail as said in the documentation. Even then, there are always small problems and overall it can be a very cumbersome process.

We found two sub-families of tools in this category, following how the analysis part is implemented. The first one is tools that are based on formal methods, which are sound mathematical foundations for proving properties on programs (e.g. model-checking and theorem proving). These tools generally have scalability problems, because of the very computation-intensive algorithms they use. This means that they can handle programs of no more than 20K lines and they can process a program of 10K lines in about 15 hours. However, their coverage is generally very good and they can find complex and obscure problems. The other family is tools based on advanced heuristics, which are clever 'rules of the thumb' for finding problems. These tools have the opposite properties: they scale well, but since their analysis is less thorough, they can only find simpler problems and coverage is not as good. In general, they can handle programs of size beyond 1 million lines of code and they can process a program of 10K lines in about 10 minutes or less. We expected formal tools to have a lesser false-positive rate than heuristics-based ones, but to our surprise that was not the case. One can generally expect a 30% false-positive rate (or less) when working with either family of tools. Our hypothesis is that many formal tools use adaptative algorithms that will often 'throw the towel', since doing a complete analysis would take too long. In other words, partial formal analysis seems to be no better than advanced heuristics for false-positives.

Static tools share a common limitation that we call the 'black box problem'. For a good verification to take place, every operation the program performs should be visible to the verifier, otherwise the verification process cannot be sound. Static source code analyzers can only 'see' source code, so when a program calls a function in a compiled library, or a program written in another programming language, something is missing. In other words, static analyzers work better on programs written in only one programming language where everything is source code. The worst case would be web applications, which are generally a mix of many programs made with different languages and platforms.

We also found that the quality of the diagnostic is very important when reviewing the problems found, especially because of false positives. A good diagnostic with a simulated execution trace allows a reviewer to quickly identify the cause of the problem and dismiss it if it does not make sense (false positive). Some tools have a very explicit and complete diagnostic, while others only tell that there is a problem around a program location. On a related subject, we found that the quality of the documentation is also very important, because many problems detected are not well known. A good description of problems and ways to correct them can save a lot of time and effort.

Contrary to our initial belief, we found that these tools are best used by the programmer himself on his own code during the development process. Since he knows his code well, he is in a better position to understand the diagnostic quickly, evaluate the impacts of the corrections that are necessary, and most important of all learn from his mistakes. Finding and correcting bugs in other people's code is a much harder task than we thought it would be and being familiar with the code creates a huge advantage.

Finally, we believe that these complex tools require quite a bit of training if one wants to use them correctly and efficiently. A good understanding of the many checkers available

and their options is a must, as are the many tricks to work with non-standard source code and buggy makefiles.

3.4.1 Boon

Description: Boon is an automated tool for finding buffer overrun vulnerabilities in C source code. Far from being an easy to use tool, it is command-line based and should work under various UNIX-like operating systems. The tool is based on David Wagner's Ph.D. dissertation and works by establishing a series of constraints that are then solved by the SML/NJ compiler.

Buzz: The tool can be compiled on most major platforms, including Cygwin. Fortunately, the required SML/NJ compiler is also available for most major platforms. Boon works better on memory buffers that are statically allocated on the stack than it does on dynamically allocated buffers on the heap. After testing various examples, the tool performed well. However, the output information is often vague. This is because in order to find the overflows the tool must generalize and thus merges some statements together in order to achieve its goal. Thus, it will under most conditions find the overflows; however, it will not be able to specifically say at which particular point in the source code the overflow occurs. Rather, it gives the name and location of the function in fault.

Status: This project is no longer under active development. It was developed by David Wagner as a part of his research project for his Ph.D. dissertation. His work was sponsored by the National Science Foundation. It is currently at version 1.0 and it was released in July 2002. Portions of its capabilities come from the BANE project.

Web: http://www.cs.berkeley.edu/~daw/boon/

Licence: BSD-like.

3.4.2 PolySpace for C/C++

Description: PolySpace for C++ is a tool based on formal methods, namely on abstract interpretation by Cousot. Because of this, it needs a lot of computing power and memory. Its thorough analysis is also not very scalable, limiting the size of programs it can handle to about 20K lines. However, PolySpace for C++ can do things other tools cannot. For example, it is very good at identifying conditions that are always true or false, allowing it to statically detect the throwing of runtime exceptions and endless loops. Its excellent pointer analysis also performs well at finding problems such as the dereference of null pointers and overflows or underflows while reading or writing to memory.

Buzz: Only the Windows version was evaluated, but the reader should know that the Linux version is supposed to be faster and use less memory. The diagnostic produced by the tool is shown in a nice graphical interface, but the information is incomplete and often not very helpful in identifying the cause of the problem. The documentation is also not very good and looks like something that was written at the last minute. The tool is strict on the

programs it will accept: almost no Microsoft or GCC extension to C/C++ is supported, which is a serious limitation to the verification of real-world programs. Also, PolySpace does not integrate well with makefiles or Microsoft Visual Studio projects.

Status: The current version is 2.5.4.

Web: http://www.polyspace.com/

Licence: Commercial.

3.4.3 Coverity Prevent

Description: Coverity Prevent is a tool based on advanced heuristics that first started as an academic project and then became a commercial product. The main selling point of this tool is its use of clever statistical analysis for finding ad hoc problems without a specification. It is very fast and scalable, being able to analyze programs of over 500K lines of code without any problem. It offers a very good integration with makefiles, but not much for Microsoft Visual Studio projects. With an add-on called Coverity Extend, it is possible to write new checkers to detect new problems.

Buzz: Coverity Prevent gives an excellent diagnostic with a simulated execution trace that is very helpful for the understanding and correction of problems. Its documentation is also excellent, giving a lot of details on the problems the tool can find, why this is a problem and how to correct it. Coverity uses the EDG compiler front-end, which is very good at handling C++ extensions from other major compilers. However, its command-line and web interface is a bit crude and slow, but functional.

Status: The product is evolving rapidly, with many new functionalities at each release. The current version is 3.0.

Web: http://coverity.com/

Licence: Commercial.

3.4.4 GrammaTech CodeSonar

Description: CodeSonar is a static source code verifier based on research conducted at the University of Wisconsin. It is sold as an add-on to CodeSurfer, a source code navigation and understanding program that does a very good pointer analysis, allowing it to see the relations between every part of the program. CodeSonar detects a limited set of problems and focuses on problems related to memory and pointer bugs.

Buzz: On Windows, CodeSonar integrates nicely with many IDEs and compilers by intercepting syscalls made to them. Compared to other similar tools, CodeSonar is a simpler program that is easy to use. It does not offer as much functionalities, but it is very good at what it does. Its detection performance and handling of C/C++ extensions is better than

average and it is very fast. The report uses colored syntax to help diagnose the problem, which is good but a simulated execution trace, as does Coverity, would have been better.

Status: The version evaluated was a prototype of 1.0. The current version is 2.0.

Web: http://www.grammatech.com

Licence: Commercial.

3.4.5 Klocwork K7

Description: Klocwork K7 is a static source code verifier for C/C++ and Java. It started as an internal project at Nortel in the 90s and later became an independent company. K7 is a bundle of several applications doing source code analysis for many purposes: security vulnerabilities discovery, architecture recovery and understanding, software metrics, etc. K7 can look for a very wide range of problems, from critical security vulnerabilities to coding style imperfections. New checkers and metrics can be created by using a well-defined and simple to use API.

Buzz: K7 integrates nicely with makefiles, Microsoft Visual Studio Projects and other major IDEs. Its truly excellent documentation covers in detail all the problems the tool can find and be of great help while reviewing bugs found. The graphical interface is by far the best of the category, mostly because it is well optimized to support the tasks related to code reviews. Reporting options are also excellent, offering a wide range of metrics to the managers to inform them where most of the problems are, what are the trends (is the software getting better or worse with time?), etc. K7's detection performance was better than any other tool, especially on buggy 'spaghetti' code and it is also very fast. C/C++ extensions were less of a problem than with other tools. The diagnostic quality is good, but a simulated execution trace would have been better. All in all, this is a very mature product and it shows in many ways.

Status: The current version is 7.1.2.

Web: http://www.klocwork.com/

Licence: Commercial.

3.4.6 Berkeley Lazy Abstraction Software Verification Tool (BLAST)

Description: BLAST is a software model checker for C programs. The goal of BLAST is to be able to check that software satisfies behavioral properties of the interfaces it uses. BLAST uses counterexample-driven automatic abstraction refinement to construct an abstract model which is model-checked for safety properties. The abstraction is constructed on-the-fly, and only to the required precision.

Buzz: In order to work, BLAST needs the appropriate Simplify theorem prover. Luckily, there are precompiled versions available for Win32, Linux, Solaris, and DEC Tru64 UNIX.

A nice feature is that there is now an Eclipse plug-in for those using Eclipse as their IDE. While Blast appears to compile under most modern UNIX-like operating systems, the same cannot be said for the Simplify theorem prover. Currently, precompiled versions exist only for the four above-mentioned operating systems. Blast can be used either from the command line or from its GUI. Many advanced C programmers are aware that it is possible to use the assert() function for simple checking of variables and conditions. Generally, these assertions can only be tested at runtime; however, using Blast it becomes possible to test these assertions statically during compile time. In general, Blast is a very powerful program that is complex to use. However, the user documentation is well written and provides several good examples to get the user up and running. In addition, the group's research papers are a very good starting point for better understanding the concepts behind Blast and the algorithms it uses.

Status: This project is still under active development. It is currently at version 2.0.

Web: http://mtc.epfl.ch/software-tools/blast/

Licence: BSD.

3.4.7 Modular Analysis of proGrams In C (MAGIC)

Description: Magic is an automated verification tool used to ascertain whether a specific implementation conforms to its specification. It was designed to work on C source code. Magic is a multiplatform application that can be compiled on various UNIX-like operating systems. However, Magic requires that the users write their own specification and that it conforms to a specific model, about which the documentation is hard to find. Unfortunately, this project, while having received good reviews, has weak technical documentation and its results cannot be reproduced or even tested by our group due to the inability of the program to run correctly.

Buzz: While compiling your own version, be sure to set the correct system variables for your particular installation (i.e. MAGICDIR, MAGICROOT, OSTYPE). Once the variables are set, it should compile on just about any UNIX-like platform with GCC, including Cygwin. During our tests, we tried to compile it under RedHat Linux 9, Solaris, and Cygwin. Although each compilation was successful, the application was not usable because it continued to give fatal errors at application startup. It appears that some modifications would have to be made to the source code. Even using the precompiled versions for RedHat 9.0 and Win32 was not successful.

Status: This project is still under active development and is currently at version 1.0. It is a project currently being undertaken by the Model Checking Group at Carnegie Mellon. The group has released many documents over the last couple of years on model checking and automated verification. While it cannot be determined exactly when the group started working on the application, the first public version, 0.1 was released in July 2003.

Web: http://www-2.cs.cmu.edu/~chaki/magic/

Licence: Unknown.

3.4.8 MOdelchecking Programs for Security properties (MOPS)

Description: MOPS is a verification tool that finds security bugs in C programs. Unlike other tools of the same kind, MOPS does not check source code for errors according to a set of predefined rules, but rather the analysts have to write their own rules and then instruct MOPS to verify if the program respects these rules or not.

Buzz: Interestingly enough, this project is a combination of both C and Java. The documentation explains very briefly how to write your own rules for detecting the presence of security bugs or the lack of them. These rule are stored in FSA and MFSA files. Once MOPS is invoked, it will perform a trace of the source code according to the rules that have been defined. This project is especially well suited to those who, after trying many of the other static analysis tools, have been disappointed by their ability to detect security problems, such as external systems granting unnecessary permissions to the program (i.e. having a setuid root bit activated). However, this program is not for the uninitiated and requires that the analyst have a very solid knowledge of the C programming language and its related problems. As a final note, a pre-built security database of possible bugs would have been greatly appreciated with the distribution of MOPS.

Status: This project is no longer under active development. It was developed by Hao Chen and David Wagner who were at the University of Berkeley, California. It is currently at version 0.9.1. It cannot be determined when the project was first developed. The current version was released October 2002.

Web: http://www.cs.berkeley.edu/~daw/mops/

Licence: BSD.

3.5 Runtime Testers

These tools try to detect errors while the program is running, by instrumenting it with checks that verify if it is in a correct state at some point. Some runtime testers instrument object code, but from our experiments, we believe that the best performing ones are those instrumenting source code.

Being generally similar to a debugger, they are considered easier to use than static analyzers, over which they have many advantages. First, their precision is supposedly perfect (no false positive or negative), because they observe a real execution, not an abstracted model of the program like static tools do. Next, there is no limit to the size of programs they can handle, since in their case verification time equals execution time. However, the coverage depends on the tests performed by the analyst and a lot of tests may be needed to achieve a coverage similar to what static tools can do. The diagnostic they give is also very precise and somewhat low-level, again a lot like a debugger. Another important advantage is that they do not suffer from the 'black box problem', as static tools do. Since they observe a real

program execution, every operation is visible and returned values from external programs or scripts (hybrid systems) are always available.

Nothing being perfect, runtime testers also have some drawbacks. The biggest one is that they require programs to be executable. This means that a program has to be completed before being able to be verified, so a partial verification early in the development process is very impractical. Finally, since these tools have a low-level view of the program, they are less versatile than static tools and tend to focus more on memory and pointer problems than on higher-level problems, like security.

3.5.1 Parasoft Insure++

Description: Parasoft Insure++ is an automated runtime tester that detects C/C++ errors such as memory corruption and pointer errors. During compilation, Insure++ inserts test and analysis functions around every source code line. Then, at runtime, it checks each data value and memory reference to verify consistency and correctness. Insure++ also does a little bit of static analysis, mostly to detect format strings problems.

Buzz: Insure++ integrates very well with Microsoft Visual Studio and makefiles in general. It also integrates with other Parasoft products, but this has not been evaluated. Insure++ detection performance is excellent, with no false positive or negative, as was expected for a tool of this kind. The diagnostic quality is very good and detailed, but we found the report to be a bit dense and hard to read. Also, it does not offer advanced functionalities for supporting the code review process, as other tools do. Finally, it is important to mention that Insure++ is very cheap compared to many static tools offering similar value.

Status: The current version is 7.0.

Web: http://www.parasoft.com

Licence: Commercial.

3.5.2 GNU Checker

Description: Checker was originally based on the GNU Malloc() program. Checker is a program that is able to find memory problems that are caused by incorrect free () or realloc () calls using a pointer which have already been freed, or when free () or malloc() have been called on a pointer which has not been obtained from a free (), malloc(), or realloc (). In short, it makes sure that when memory is allocated or freed, it is done in a correct and safe manner.

Buzz: While there are many other programs available that can check the integrity of memory calls inside of a program, this program functions only at runtime. Checker must first be compiled for a specific platform and only Linux and Solaris are supported. The source code of the program to verify has to be instrumented by calling Checker, which will also compile it. The program is then run normally and if any problem occurs, it will be

printed to stdout. A very nice feature is that when the program outputs information to stdout, it will print the offending source code line numbers. It should also work with both C and C++ code.

Status: This project is no longer under active development and has not been since august 1998. It was developed by Tristan Gingold, maintained and development aided by Ben Pfaff. The current release of the program is 0.9.9.1. The first public release was in 1994 and was version 0.1e-7.

Web: http://www.gnu.org/software/checker/

Licence: GNU GPL.

3.5.3 ElectricFence

Description: Electric Fence is similar in its abilities to catch memory errors to Checker. However, Electric Fence is particularly well suited to catching memory read or written out-of-bounds errors. It too, like Memwatch and Checker, instruments the source code to detect problems at runtime. It uses its own virtual memory address pages in order to catch memory errors and it is able to report the exact location in memory where the error occurred.

Buzz: Unlike Checker or Memwatch, ElectricFence is actually a library that one compiles with his source code. When calling the compiler to compile the program to be verified, one must also include the '-lefence' option in order to link the library with the source code. It works essentially the same way as Checker. However, Checker does detect out-of-bounds memory errors. For each malloc() call used for memory reservation, two pages of virtual memory are required, which usually uses up to 16 KB of extra memory per malloc() call. This of course will vary according to operating systems. Currently, it can be ported to Solaris, HP-UX, IRIX, AIX, OSF, Linux, and other i386 UNIX SVR4 compatible platforms.

Status: While there appear to be many versions of the project in compiled form for many different platforms, it is difficult to determine when the last platform independent form of the source code was available. It seems to have been first published in 1995 and is now at version 2.0.5. It was developed by Bruce Perens. It appears that there are versions that go up 2.2.2, but their source code cannot be found.

Web: http://perens.com/FreeSoftware/

Licence: GNU GPL.

3.5.4 MemWatch

Description: Memwatch can detect memory leaks, buffer underflows and overflows, plus some pointer problems. However, it was not designed to work with C++ classes and therefore, it can only be used with C code. Like ElectricFence, it is a library that has to be linked with the program to work. It is portable to most platforms.

Buzz: In order to use Memwatch, one has to modify each source code file to add the line '#include memwatch.h' and compile it with special directives. This can be cumbersome for large projects having many source code files.

Status: While it cannot be determined exactly when the project started, it has been registered with SourceForge since May 2001. The author of the program is Johan Lindh. The project is currently at version 2.71 and it was released in 2003. It is uncertain if the project is still under active development.

Web: http://memwatch.sourceforge.net/

Licence: GNU GPL.

4 Evaluation

We wanted to know how well the defect finding tools perform in real test cases and also to see if they really are as good as claimed. To answer these questions, we tested the five most promising tools (Coverity Prevent, PolySpace for C++, Grammatech CodeSonar, Klocwork K7, and Parasoft Insure++) in two ways. First, over a real code in production that, to our knowledge, worked well but was a bit buggy, and then over many small ad hoc pieces of code containing specific programming defects, which we called synthetic tests.

4.1 Methodology

Runtime testers detect errors, while static analyzers detect defects. To allow for a comparison, defects were 'transformed' into their corresponding error type. To do so, every defect found was analyzed manually to see what kind of error it would create. The location of errors was also matched with defects for more precision.

We also applied the following rules to decide if a tool had detected a problem or not:

- If a true problem was found with a correct diagnostic: TRUE POSITIVE
- If a true problem was found with an incorrect diagnostic: TRUE POSITIVE
- If a true problem was not found: FALSE NEGATIVE
- If a false problem was found: FALSE POSITIVE

We decided to consider a problem found with an incorrect diagnostic as a true positive, because some tools had a very poor diagnostic quality and they would have been disadvantaged even if they found the problem. As long as the problem was found in the correct location, it was ok.

4.2 Synthetic tests

A test framework with a C++ class for every kind of defect was created and integrated into a small, high-quality open-source application built with the Microsoft Foundation Classes (MFC) framework. Defects were called from the main() of the application, after initialization but before the program started to answer user queries. Defects that would lead to a program crash or hang were deactivated for Insure++, since we wanted to run all tests in a single pass.

Applications built with MFC do not have a concrete main(). Instead, the program starts when the application object is created. This was a problem for PolySpace, which cannot handle that kind of main(). Therefore, it had to be used in a class-by-class analysis mode instead of a whole-program analysis. Our defects were thus designed to be detectable even without full inter-procedural analysis.

4.2.1 Results

The results of our synthetic tests are shown in Figures 6, 7, 8, 9, and 10. No tool tries to detect every kind of defects or errors. However, all together, the tools detected all but four problems. There were no false positives, except for PolySpace that only had a few. All tools focus more or less on the same kind of problems, except PolySpace, with its thorough analysis, that was able to detect arithmetic and cast faults.

4.3 Production Code Tests

The code used was a numerical analysis application of about 10K lines that had been in production for many years. The code was functional but a bit buggy and not very well designed (that is, a 'C+' design – everything in one big class). As an example, we found many cut-and-pasted segments of code that could have been refactored into a method.

Since the program was pretty straightforward (it reads a file and displays results), with a call graph more or less like a straight line, we only tested one execution. This gave us sufficient coverage to allow a comparison between runtime testers and static analyzers.

4.3.1 Results

Since the synthetic tests gave us no incredibly great or poor results (i.e. all tools performed reasonably well), we were surprised by the results of production code tests. All tools performed very badly, except Klocwork K7 and Parasoft Insure++. We tried the tests many times, with different configurations, but we were consistently unable to obtain good results. It seems that low-quality, 'spagnetti' code using pointers heavily can be very hard to analyze for most tools, or something was wrong with our experiments. Because of these strange results, we do not provide details and numbers, since we do not know if they are correct. It is important to mention, however, that Klocwork K7 and Parasoft Insure++ always performed consistently, whatever the code we tried to verify.

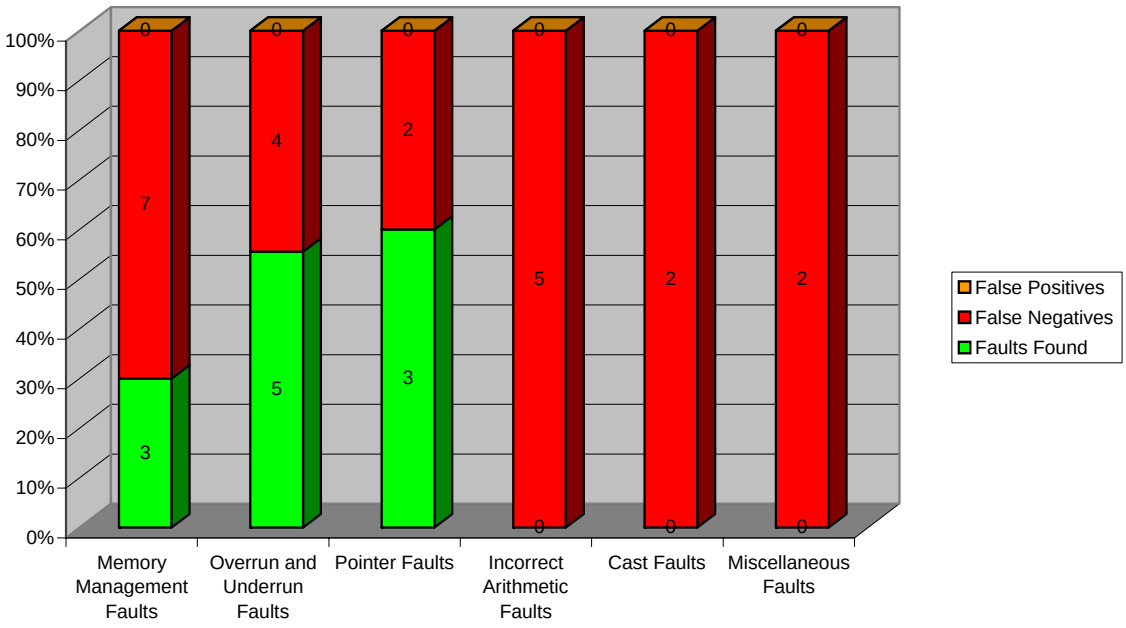

Figure 6: *Coverity Prevent Results*

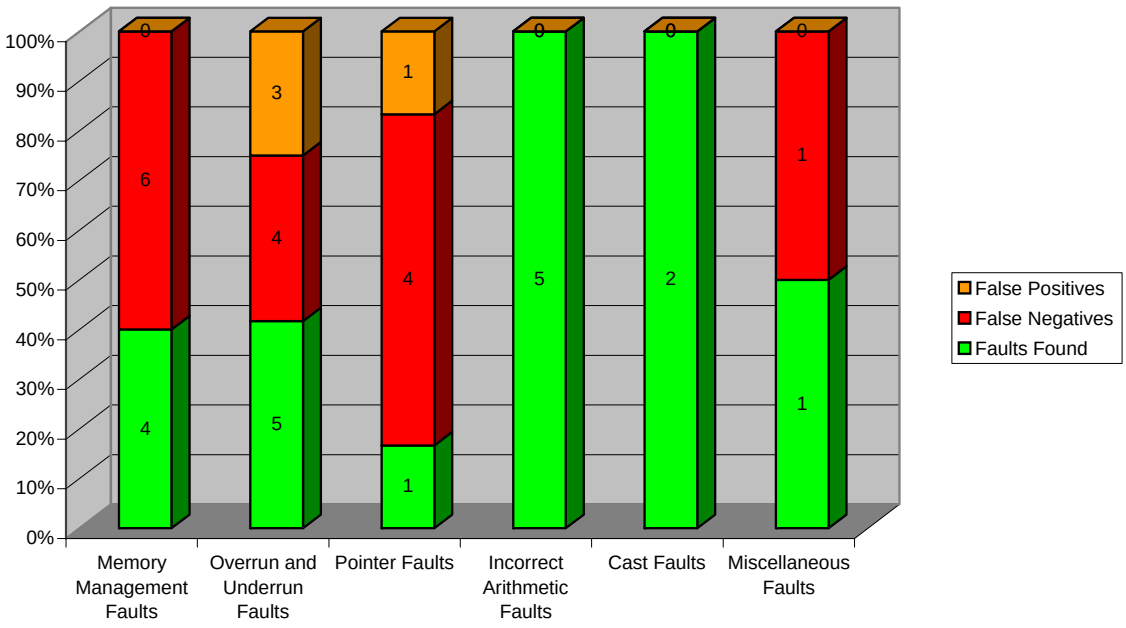

Figure 7: *PolySpace for C++ Results*

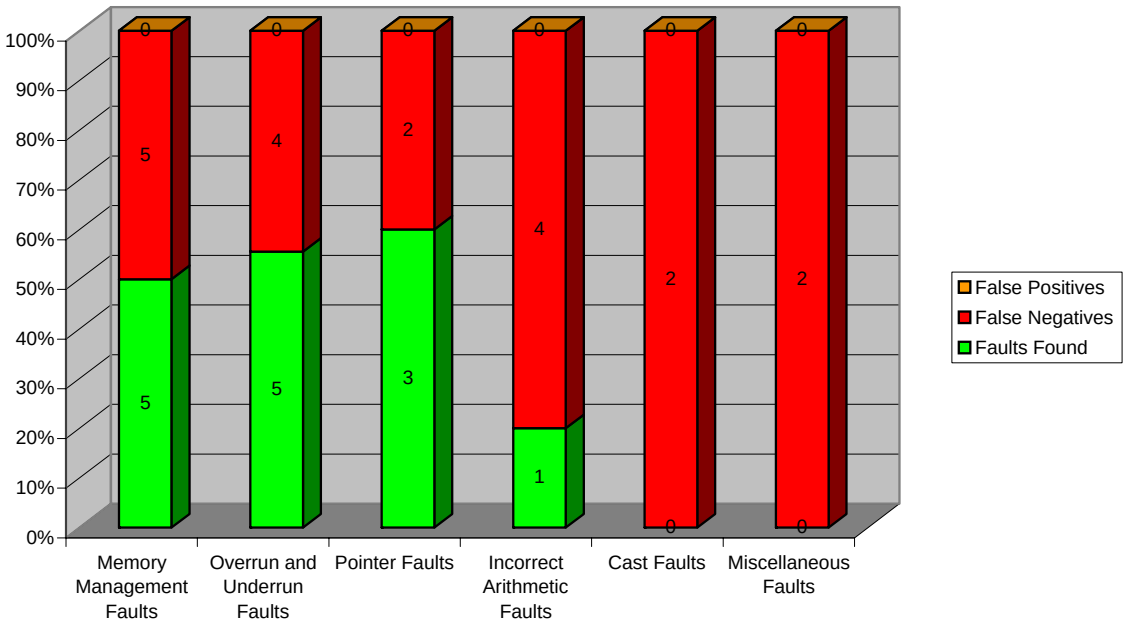

Figure 8: *GrammaTech CodeSonar Results*

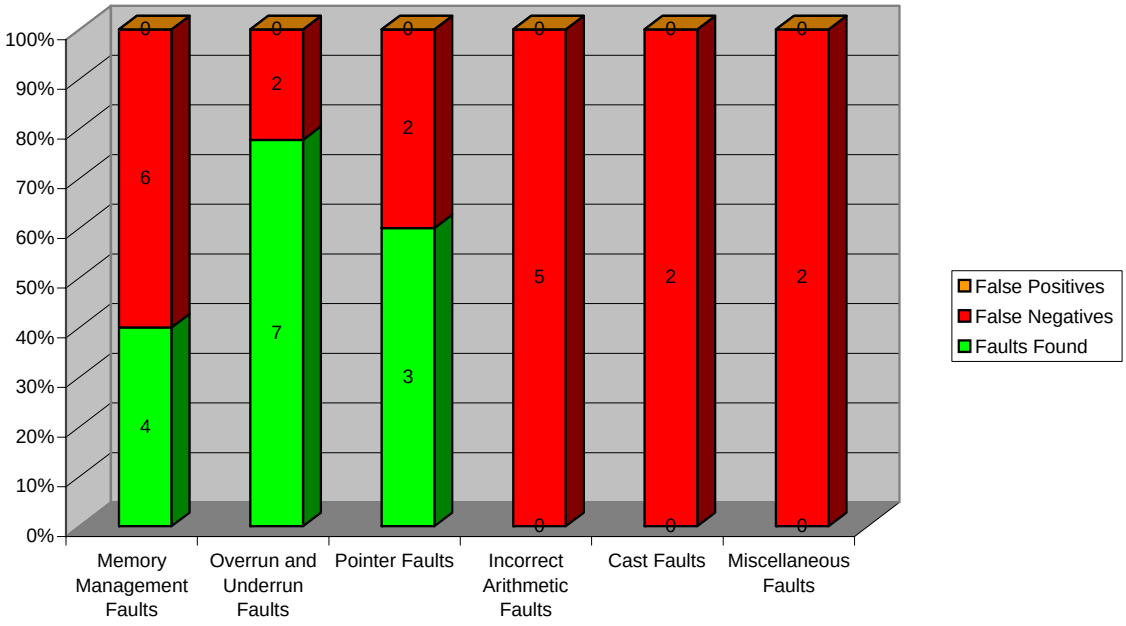

Figure 9: *Klocwork K7 Results*

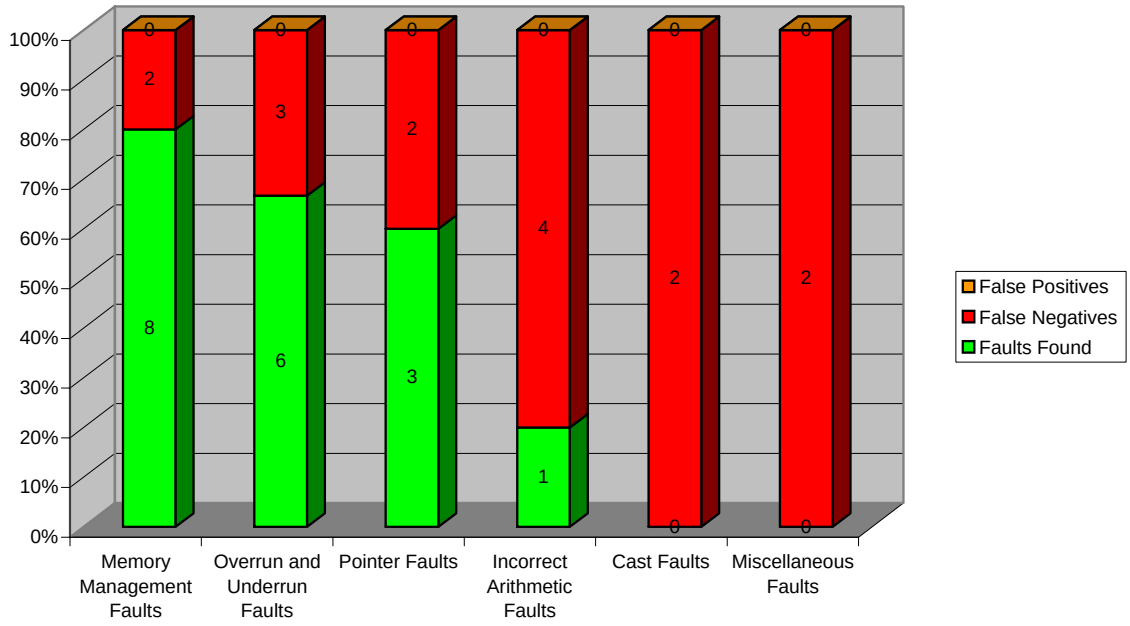

Figure 10: *Parasoft Insure++ Results*

Conclusion

Security problems generally do not come from the failure of security mechanisms. The failures occurs at a lower level, because of program sanity problems. C/C++ are especially problematic because they enforce almost no restriction on the execution of programs and they are prone to vulnerabilities with serious consequences, such as buffer overflows.

Verifying C/C++ programs is a huge challenge. These languages are very difficult to analyze because of many undefined or non-standard semantics, pointer arithmetic, compiler-specific extensions to the language, etc.

There are many different and unrelated techniques to verify software, each having its pros and cons. Scalability and precision are difficult to get together, and it is often at the expense of coverage. It seems one cannot have it all. Also, finding defects does not correct them and no tool evaluated can suggest a fix to correct a problem. Sometimes, it is not even clear why there is a problem. Some products, like Insure and Coverity, are very good at explaining why there is a problem, but others put a lot less effort. Finding complex or higher-level problems, like design problems, is still way over the head of these tools.

The diagnostic quality is of high importance if there are a lot of false positives. It can be very hard to understand the cause of an error and if one is not even sure that an error is present. Support for change impact analysis would be very useful to help the analyst make corrections to defective programs.

Recommendations

Our main recommendation is to use better languages and platforms, such as Java, C#, Ada, or any other modern programming language. Patching the problems is a solution, but to bypass it entirely by using better languages and platforms is a better one. Of course, this is not always compatible with requirements and the expertise to use these better languages and platforms may be lacking. Also, C and C++ are still often the best solution for some uses, like realtime high performance systems and embedded platforms. When the use of C/C++ is mandatory, we recommend to restrict the language use to a safer subset, such as MISRA or JSF++.

Library replacements and program hardeners are good pervasive solutions to reduce the risk of errors with minimal effort. However, their effectiveness is limited and they should be used in a 'defense in depth' perspective (i.e. with other methods). For a quick and dirty verification, static analyzers based on program syntax will do a good job. However, if one is serious about program verification, static analyzers based on program semantics and runtime testers should be used. The following presents the best usage scenario for the best performing tools we evaluated.

Klocwork K7, Coverity Prevent

The best usage scenario for these tools is when the whole application needs to be analyzed and it is compiled using a working makefile. The application code size can be over 500K lines of C++ without problem. Coverity and Klocwork have many good points: very good integration with makefiles, can read code that contains compiler-specific extensions from almost every major compiler in the industry, very scalable, and good diagnostic to help understand problems.

PolySpace for C++

The best usage scenario for PolySpace for C++ is to analyze small segments of critical code in applications where runtime exceptions should never happen. The application code size must stay under 20K lines of C++. It uses a very thorough analysis based on abstract interpretation, with which it can detect runtime errors statically. It has a nice graphical interface, especially the *Viewer* module that is used to analyze the report and navigate in the source code. However, it lacks a good diagnostic because sometimes, it is impossible to understand the defect found. Moreover, it is sometimes necessary to modify the analyzed source code to have a correct model (e.g., reactive applications wait for user inputs, so one has to simulate them to analyze the reactions). Its analysis stops after critical errors and the command to override this behavior is undocumented, and finally, it is slow and memory hungry, but this is expected with such a thorough analysis.

Parasoft Insure++

The best usage scenario for Parasoft Insure++ is to test hybrid systems based on many heterogeneous components. To consider code coverage, it should always be integrated into test case harnesses. Since Insure++ is a dynamic tool, there is no limit to the application code size and bad quality code has no effect on detection performance. Insure++ has a very good diagnostic with call stack and memory diagrams that show exactly what was overwritten. However, test cases have to be carefully specified with a good coverage strategy.

References

[1] Avizienis, Algirdas, Laprie, Jean-Claude, Randell, Brian, and Landwehr, Carl (2004). Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, pp. 11–30. Read the Document.

[2] Raynal, Frederic, Blaess, Christophe, and Grenier, Christophe (2001). Avoiding security holes when developing an application - 2: memory, stack and functions, shellcode. Technical Report. Read the Document.

[3] Raynal, Frederic, Blaess, Christophe, and Grenier, Christophe (2001). Avoiding security holes when developing an application - Part 3: buffer overflows. Technical Report. Read the Document.

[4] Lacroix, Patrice (2003). Buffer Overflows and Format String Vulnerabilities. Technical Report. Laval University. Read the Document.

[5] Fayolle, Pierre-Alain and Glaume, Vincent (2002). A Buffer Overflow Study Attacks & Defenses. Technical Report. ENSEIRB. Read the Document.

[6] Mixter. Writing buffer overflow exploits - a tutorial for beginners. Technical Report. Read the Document.

[7] One, Aleph. Smashing The Stack For Fun And Profit. *Phrack*, Vol. 7. Read the Document.

[8] Conover, Matt and w00w00 Security Team (1999). w00w00 on Heap Overflows. *Phrack*. Read the Document.

[9] Wheeler, David A. (2003). Secure Programming for Linux and Unix HOWTO. Technical Report. Read the Document.

[10] Thuemmel, Andreas (2001). Analysis of Format String Bugs. Technical Report. Read the Document.

[11] Raynal, Frederic, Blaess, Christophe, and Grenier, Christophe (2001). Avoiding security holes when developing an application - 4: format strings. Technical Report. Read the Document.

[12] Raynal, Frederic (2001). How to Remotely and Automatically Exploit a Format Bug. Technical Report. Read the Document.

[13] scut and team teso (2001). Exploiting Format String Vulnerabilities. Technical Report. Read the Document.

[14] Kiriansky, Vladimir L. (2003). Secure Execution Environment via Program Shepherding. Master's thesis. Massachusetts Institute of Technology. Read the Document.

[15] Bulba and Kil3r (2000). Bypassing Stackguard and Stackshield. *Phrack Magazine*, Vol. 10. Read the Document.

[16] Berger, Emery D. and Zorn, Benjamin G. (2006). DieHard: Probabilistic Memory Safety for Unsafe Languages. In *PLDI'06*. Read the Document.

[17] Wilander, John and Kamkar, Mariam (2002). A Comparison of Publicly Available Tools for Static Intrusion Prevention. In *7th Nordic Workshop on Secure IT Systems*. Read the Document.

[18] Heffley, Jon and Meunier, Pascal (2004). Can Source Code Auditing Software Identify Common Vulnerabilities and Be Used to Evaluate Software Security?. In *37th Hawaii International Conference on System Sciences*. Read the Document.

[19] La, Thien (2002). Secure Software Development and Code Analysis Tools. Technical Report. SANS Institute. Read the Document.

[20] Evans, David and Larochelle, David (2002). Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software*, pp. 42–51. Read the Document.

[21] Viega, John, Bloch, J. T., Kohno, Tadayoshi, and McGraw, Gary. ITS4: A Static Vulnerability Scanner for C and C++ Code. Technical Report. Reliable Software Technologies. Read the Document.

This page intentionally left blank.

Distribution list

DRDC Valcartier TR 2006-735

Internal distribution

1 Director General

3 Document Library

1 Head/Information and Knowledge Management

1 Head/Systems of Systems

1 Robert Charpentier

1 François Lemieux

1 Frédéric Michaud

1 Frédéric Painchaud

1 Martin Salois

1 Mario Couture

Total internal copies: 12

External distribution

National Defense Headquarters - 101 Colonel By Drive, Ottawa (ON) Canada K1A 0K2

1 Julien A. Bourdeau (DTA-5)

1 Sylvain J. Fleurant (DTAES 6)

Total external copies: 2

Total copies: 14

This page intentionally left blank.

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when document is classified)

1. ORIGINATOR (the name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Centre sponsoring a contractor's report, or tasking agency, are entered in section 8.)

   DRDC Valcartier
   2459 Pie-XI Blvd. Quebec, Quebec, Canada G3J 1X5

2. SECURITY CLASSIFICATION (overall security classification of the document including special warning terms if applicable).

   UNCLASSIFIED

3. TITLE (the complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S,C,R or U) in parentheses after the title).

   Practical verification & safeguard tools for C/C++

4. AUTHORS (last name, first name, middle initial)

   Michaud, F.; Carbone, R.

5. DATE OF PUBLICATION (month and year of publication of document)	6a. NO. OF PAGES (total containing information. Include Annexes, Appendices, etc).	6b. NO. OF REFS (total cited in document)
November 2007	70	21

7. DESCRIPTIVE NOTES (the category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered).

   Technical Report

8. SPONSORING ACTIVITY (the name of the department project office or laboratory sponsoring the research and development. Include address).

   DRDC Valcartier
   2459 Pie-XI Blvd. Quebec, Quebec, Canada G3J 1X5

9a. PROJECT NO. (the applicable research and development project number under which the document was written. Specify whether project).	9b. GRANT OR CONTRACT NO. (if appropriate, the applicable number under which the document was written).
15BP01	

10a. ORIGINATOR'S DOCUMENT NUMBER (the official document number by which the document is identified by the originating activity. This number must be unique.)	10b. OTHER DOCUMENT NOs. (Any other numbers which may be assigned this document either by the originator or by the sponsor.)
DRDC Valcartier TR 2006-735	

11. DOCUMENT AVAILABILITY (any limitations on further dissemination of the document, other than those imposed by security classification)

    (X) Unlimited distribution
    ( ) Defence departments and defence contractors; further distribution only as approved
    ( ) Defence departments and Canadian defence contractors; further distribution only as approved
    ( ) Government departments and agencies; further distribution only as approved
    ( ) Defence departments; further distribution only as approved
    ( ) Other (please specify):

12. DOCUMENT ANNOUNCEMENT (any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution beyond the audience specified in (11) is possible, a wider announcement audience may be selected).

13. ABSTRACT (a brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual).

This document is the final report of an activity that took place in 2005-2006. The goal of this project was first to identify common software defects related to the use of the C and C++ programming languages. Errors and vulnerabilities created by these defects were also investigated, so that meaningful test cases could be created for the evaluation of best-of-breed automatic verification tools. Finally, when relevant, best practices were inferred from our experiments with these tools.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus. e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus-identified. If it not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title).

Software, Verification, Validation, Vulnerabilities, Flaws, Defects, C, C++

Defence R&D Canada

Canada's Leader in Defence
and National Security
Science and Technology

R & D pour la défense Canada

Chef de file au Canada en matière
de science et de technologie pour
la défense et la sécurité nationale

WWW.drdc-rddc.gc.ca